Freescale Semiconductor, Inc.

OSEKturbo Design Tool for Deterministic Scheduling v.1.1

User's Manual

Revised <October 2003>

metrowerks

For More Information: www.freescale.com

© 2003 MOTOROLA, ALL RIGHTS RESERVED

Motorola reserves the right to make changes without further notice to any products herein to improve reliability, function or design. Motorola does not assume any liability arising out of the application or use of any product or circuit described herein; neither does it convey any license under its patent rights nor the rights of others. Motorola products are not designed, intended, or authorized for use as components in systems intended for surgical implant into the body, or other applications intended to support or sustain life, or for any other application in which the failure of the Motorola product could create a situation where personal injury or death may occur. Should Buyer purchase or use Motorola products for any such unintended or unauthorized application, Buyer shall indemnify and hold Motorola and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claim alleges that Motorola was negligent regarding the design or manufacture of the part. Motorola and Ⓜ are registered trademarks of Motorola, Inc. Motorola, Inc. is an Equal Employment Opportunity / Affirmative Action Employer.

Legal Notices

The information in this document has been carefully checked and is believed to be entirely reliable, however, no responsibility is assumed for inaccuracies. Furthermore, Motorola reserves the right to make changes to any products herein to improve reliability, function or design. Motorola does not assume liability arising out of the application or use of any product or circuit described herein; neither does it convey any license under its patent rights or the rights of others.

The software described in this document is furnished under a license agreement. The software may be used or copied only in accordance with the terms of the agreement.

No part of this publication may be reproduced or transmitted in any form or by any means - graphic, electronic, electrical, mechanical, chemical, including photocopying, recording in any medium, taping, by any computer or information storage retrieval systems, etc., without prior permissions in writing from Motorola Inc.

Permission is granted to reproduce and transmit the Problem Report Form, the Customer Satisfaction Survey, and the Registration Form to Motorola.

Important Notice to Users

While every effort has been made to ensure the accuracy of all information in this document, Motorola assumes no liability to any party for any loss or damage caused by errors or omissions or by statements of any kind in this document, its updates, supplements, or special editions, whether such errors are omissions or statements resulting from negligence, accident, or any other cause. Motorola further assumes no liability arising out of the application or use of any product or system described herein; nor any liability for incidental or consequential damages arising from the use of this document. Motorola disclaims all warranties regarding the information contained herein, whether expressed, implied or statutory, including implied warranties of merchantability or fitness for a particular purpose.

Trademarks

OSEK / VDX is a registered trademark of Siemens AG.
Metrowerks, the Metrowerks logo and CodeWarrior are registered trademarks of Metrowerks Inc.
Microsoft, MS-DOS and Windows are trademarks of Microsoft.

Contents

1 Overview 7

2 Introduction 9

Technical Reference Structure 9
Typographical Conventions 10
References 11
Definitions, Acronyms and Abbreviations 12
Technical Support Information 13

3 Schedulability Analysis Basics 15

Real-time Concept 15
Objectives of Schedulability Analysis 19
Main Goal of Schedulability Analysis 19
Modeling Approach in Schedulability Analysis 19
Schedulability Analysis and Testing 20
Schedulability Analysis in the Development Process 21
What Schedulability Analysis Does 22
What Schedulability Analysis Does Not Do. 22
Basic Methods of Schedulability Analysis 23
Advanced Methods of Schedulability Analysis 24

4 Tool Architecture 27

DS-Design Tool Components 27
OSEK Builder 29
OIL File 29
Schedulability Analysis 29
Graphical Views 30

5 Getting Started 31

Application Sample. 31
Defining Application Timing Parameters 32
Starting Analysis 38
Exploring Analysis Results. 38
Changing the application timing behavior 39
GettingStarted.oil file content 41

6 Analysis of BCC Applications 43

General 44
Transactions 46
Transactions Overview 46
Application Transactions 47
Periodic and Sporadic Transactions 48
Subtransactions 49
Subtransactions Overview 49
Applications Subtransactions 50
Tasks and ISRs Sections 51
Tasks and ISRs Sections Overview 51
Task of Single Task Section 52
Sections and Execution Scenario 53
Sections and Rescheduling points. 55
Sections and Processing Levels (priorities) 57
Sections and Conditional Operators 58
Actions Summary 60
Timepoints and Checkpoints 62
Timepoints and Checkpoints Overview 62
Checking Deadlines in the Middle of Tasks and ISRs 63
Execution Paths 64
Using Paths for Dependent Conditional Operators 64
Using Paths Across Tasks and ISRs 66
Applying Computational Model 70
Building an Application Model 70
Single Periodic Task 71
Periodic Task and an ISR 84
Periodic Task, an ISR and the Resource 92
Periodic Task, an ISR, and Short Critical Section 98
Periodic Task, an ISR, and GoTo actions 103
Periodic Task, an ISR, and Paths 110
Tuning an Application 117
General Recommendations 117
Rethinking Hardness of Deadlines 119
Increasing Speed of Processor 119
Using Timescale for Periodic Tasks 120
Making Periods Harmonic 124

Restructuring subtransactions 124
Restructuring Critical Sections 130
Changing Scheduling Policy 130
Employing OSEKturbo Internal Resources 134
Using Paths 141
Limitations for the Application Structure 142
Dictionary of DS-Design Tool 142

7 Analysis of ECC Applications 145
General 145
Particularities of Waiting State 145
Applying Computational Model 146
Extended Task with Single Event and an ISR 146
Extended Task with Two Events 151
Tuning an Application 157
Limitations for Application Structure 158

8 Advanced Features 159
General 159
Internal Resources 159
Using text views 160
Fine Tuning of Applications 161
Batch Mode of Analysis 162
Dealing with Operating System overhead 163

9 Troubleshooting 165
Evaluation of Timing Parameters 165
Eliminating Wrong Application Structure 166
Looking at Intermediate Data Files 166
Computation Issues 166
Full Utilization of Processor 167
Blocking Pessimism 167
Known Problems 170

A Sample Application 171
Description 171
Source Files 171
Results of Analysis 177

B Error Messages **181**

List of System Generator Messages 181

. 188

C OIL Language Quick Reference **189**

OS Object 190

TASK Object 194

ISR Object 199

RESOURCE Object 203

EVENT Object 204

COUNTER Object 204

ALARM Object 204

MESSAGE Object 205

APPMODE Object 205

COM Object 205

NM Object 205

Index **207**

1

Overview

OSEKturbo Design Tool for Deterministic Scheduling v.1.1 (DS-Design Tool) is the new product in family of OSEK software products from Metrowerks. DS-Design Tool provides the developer of OSEKturbo applications with powerful and reliable means of schedulability analysis of applications.

DS-Design Tool is integrated into OSEK Builder version 2.3 and is coupled with OSEKturbo Operating System.

DS-Design Tool provides the following features:

- A schedulability verdict of the application;
- The calculation of total utilization of CPU;
- The calculation of the actual values of the response times for the application deadlines;
- The response time structure (contribution of various activities to the actual response time).

The application designer inputs timing parameters of applications into DS-Design Tool tool by means of using OSEK Builder. DS-Design Tool provides the results of analysis in an easy to understand graphical format. Different views allow flexible and detailed insight of the application timing behavior. Based on the results of analysis the application designer makes the decision if changes in application timing behavior are needed in order to improve application scheduling.

DS-Design Tool should be used for real-time applications to verify that the deadlines are always met. The schedulability analysis is not the replacement of testing, nor it can be replaced by testing. Any testing may prove the correctness of the application behavior in the certain set of conditions, but it can not prove that the application meats deadlines in all possible conditions. Schedulability analysis can do that, because it uses sophisticated mathematical methods that evaluate the scheduling of application in all conditions, including worst-case scenarios, thus proving correctness of timing behavior of the application during application life time.

DS-Design Tool should be considered as an integral part of application development process, and can be applied at different phases of the development.

DS-Design Tool is based on state-of-the-art methods developed specifically for this tool. The methods are derived from known ones, but they were significantly advanced in order to provide more accurate analysis. While these methods are rather general, they were refined and adopted for OSEKturbo Operating System to cover the particularities of Operating System architecture and features it provides.

DS-Design Tool is well documented and tested. In the User's Manual architecture, features, particularities, and the scheduling analyzing techniques are described in detail with numerous examples.

2

Introduction

This chapter consists of the following sections:

- Technical Reference Structure
- Typographical Conventions
- References
- Definitions, Acronyms and Abbreviations
- Technical Support Information

Technical Reference Structure

This Technical Reference consists of the following sections:

"Overview" chapter describes what the DS-Design Tool is and highlights its basic features.

"Introduction" chapter contains a description of the Technical Reference structure, typographical conventions and the list of acronyms.

"Schedulability Analysis Basics" chapter explains the schedulability analysis concept and how it is related to OSEKturbo applications.

"Tool Architecture" chapter gives a high level description of DS-Design Tool architecture.

"Getting Started" chapter provides a quick introduction into the tool's usage for simple example of user application.

"Analysis of BCC Applications" chapter describes the usage of the tool for schedulability analysis of the applications of OSEK/VDX OS Basic Conformance Classes.

"Analysis of ECC Applications" chapter describes particularities of analysis of the applications that belong to OSEK/VDX OS Extended Conformance Classes.

"Advanced Features" chapter describes advanced features of the tool that are aimed to analyze sophisticated applications.

"Troubleshooting" chapter describes support provided to user to investigate problems that may arise when using the tool.

"Sample Application" appendix contains text, configuration, and explanation of analysis results for a sample customer application analyzed by means of using DS-Design Tool.

"Error Messages" appendix provides information about the tool's error messages.

"OIL Language Quick Reference" appendix provides information about OIL attributes used to configure schedulability.

NOTE The reader of this User's Manual might find useful to learn OSEKturbo Operating System before or simultaneously with learning DS-Design Tool.

Typographical Conventions

This Technical Reference employs the following typographical conventions:

boldface type

Bold is used for important terms, notes and warnings.

Italics

Italics are used for all names of commands, constants, routines and variables.

`Courier` font

Courier typeface is used for code examples in the text.

`Courier Italic`

Courier Italic typeface is used for terms when these are first introduced.

References

[1] Adding Time-Offsets to Schedulability Analysis, by Ken Tindell, http://www-users.cs.york.ac.uk/~jac/papers/offsets.ps

[2] Schedulability Analysis for Tasks with Static and Dynamic Offsets, by J.C. Palencia, M. Gonzalez Harbour, http://www.ctr.unican.es/publications/jcp-mgh-1998a.pdf

[3] Exploiting precedence relations in the Schedulability Analysis of Distributed Real-Time Systems, by J.C. Palencia, M. Gonzalez Harbour, http://www.ctr.unican.es/publications/jcp-mgh-1999b.pdf

[4] Real-Time Systems. Design Principles for Distributed Embedded Applications, by Hermann Kopetz, 1997, ISBN 0-7923-9894-7

[5] A Practitioner's Handbook for Real-Time Analysis, by Mark H. Klein, Thomas Ralya, Bill Pollak, Ray Obenza, 1993, ISBN 0-7923-9361-9

[6] Real-Time Systems, by Jane W.S. Liu, Prentice Hall, 1st edition (June 15, 2000), ISBN: 0-1309-9651-3

[7] C. L. Liu, James W. Layland “Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment”, Journal of the ACM, 20(1), 1973, pp 46-61

[8] OSEK / VDX System Generation OIL: OSEK Implementation Language Version 2.3, September 10th, 2001

[9] OSEK/VDX Operating System, Version 2.2 revision 1, September, 10th, 2001

[10] Scheduling Fixed-Priority Tasks with Preemption Threshold, by Yun Wang, Manas Saksena, 1999 (This work was presented in part at RTCSA'99, Hongkong, December 1999 and a brief version of this work was published in the conference proceedings of RTCSA'99 by the IEEE Computer Society Press)

[11] IEEE Standard Glossary of Software Engineering Terminology, IEEE Std 610.12-1990

[12] IEEE Standard for Information Technology—Portable Operating System Interface (POSIX®) — Part 1: System Application Program Interface (API) — Amendment d: Additional Realtime Extensions [C Language], IEEE Std 1003.1d-1999

Definitions, Acronyms and Abbreviations

API	Application Program Interface (a set of data types and functions)
BCC	Basic Conformance Class, a defined set of functionality in OSEK, for which the waiting state of tasks is not permitted
BT	Basic task (task, which never has the waiting state)
CPU	Central Processor Unit
DLL	Dynamic Link Library
DMA	Deadline Monotonic Algorithm
DS-D	Design Tool for Deterministic Scheduling
DS-Design Tool	Design Tool for Deterministic Scheduling
DS-V	Verification Tool for Deterministic Scheduling
ECC	Extended Conformance Class, a defined set of functionality in OSEK, for which the waiting state of tasks is permitted
ECU	Electronic Control Unit (similar to MCU)
ET	Extended Task (task, which may have the waiting state)
ID	Identifier, an abstract identifier of a system object
ISR	Interrupt Service Routine
MCU	Microcontroller Unit (Motorola’s microcontrollers)
N/A	Not applicable
OIL	OSEK Implementation Language
ORTI	OSEK Run Time Interface
OS	Operating System, a part of the OSEK
OSEK	Open systems and their corresponding interfaces for automotive electronics (in German)
RAM	Random Access Memory
RMA	Rate Monotonic Algorithm

ROM	Read Only Memory
SA	Schedulability Analysis
SG, SysGen	System Generator
WCET	Worst Case Execution Time

Technical Support Information

To order Motorola/Metrowerks products or literature, consult your local sales representative.

For technical support please use:

- US

  Tel : +1 512 997 4700

  Fax : +1 512 997 4901

  Email : support@metrowerks.com

- Europe

  Tel : +41 61 69 07 505

  Fax : +41 61 69 07 501

  Email : support_europe@metrowerks.com

- Web : http://www.metrowerks.com/MW/Support
- Download updates at http://www.metrowerks.com/MW/Support/Download

3

Schedulability Analysis Basics

This chapter consists of the following sections:

- Real-time Concept
- Objectives of Schedulability Analysis
- Basic Methods of Schedulability Analysis
- Advanced Methods of Schedulability Analysis

Real-time Concept

The OSEK Operating System is a real-time operating system, and it is aimed to build real-time applications. But what is real-time application or real-time system?

There are various definitions of real-time systems, and here are some of them.

IEEE Glossary includes the following definition (see **[11]** in section "References,"):

> **"***real time*. Pertaining to a system or mode of operation in which computation is performed during the actual time that an external process occurs, in order that the computation results can be used to control, monitor, or respond in a timely manner to the external process.**"**

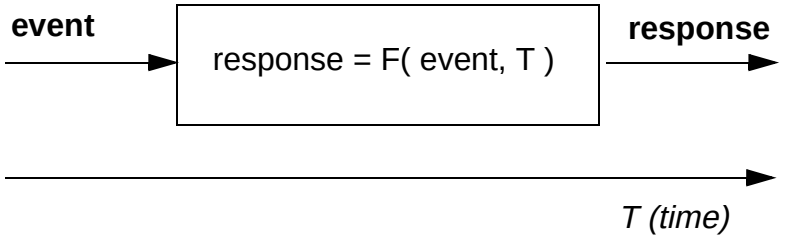
IEEE POSIX specifies *realtime in operating systems* as "the ability of the operating system to provide a required level of service in a bounded response time" (see **[12]** in section "References").

The expert in real-time system Hermann Kopetz defines real-time computer system as "a computer system in which the correctness of the

system behavior depends not only on the logical results of computations, but also on the physical instant at which these results are produced" (see **[4]** in section "References").

While all these definitions seem to be different, the common thing is the ability of the system to provide computation results, i.e. response in a **bounded response time**. In other words, response of the system is a function of event and time, as it is shown on **Figure 3.1**.

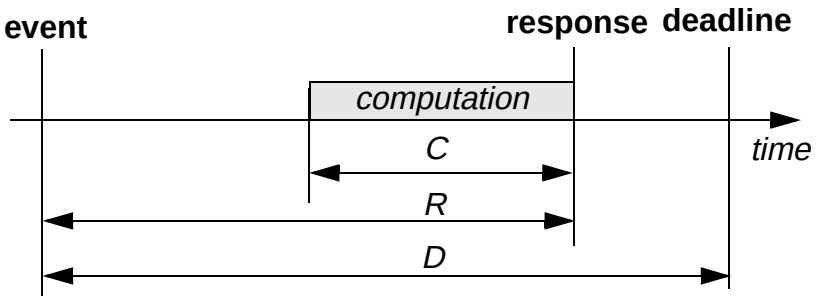
Figure 3.1 Notion of event and response



The bound of response time is called *deadline*. Generally speaking, the real-time system should provide response for the arrived event before deadline expires. Deadlines can be either **soft** or **hard**. **Figure 3.2** presents the hard deadline, and introduces the following basic attributes of event and response:

- *C*, *computation time*. This is time interval needed for processor unit to make computations and generate response;
- *R*, *response time*. This is actual response. It is measured from the time point when event arrives till the time point when response is generated;
- *D*, deadline. This is deadline for the response time. It is measured from time point when event arrives till the time point when response must be generated.

Figure 3.2 Hard deadline



C - computation time for an event
R - response time
D - deadline for an event

`Hard deadline` is a deadline which must be always met (i.e. system must generate response to the event before deadline expires). If hard deadline is missed, then system experienced failure. If *`soft deadline`* is missed, then system degrades, but does not fail. Therefore, soft deadline should be met in most cases, but may be missed occasionally.

If all or some deadlines in an application are hard deadlines, then the application is called *`hard real-time application`*. Most embedded application designed for OSEKturbo are hard real-time applications.

NOTE Even if application requires hard real-rime, it is important to distinguish soft and hard deadlines within the application. If the deadline of some responses may be missed without fatal consequences, then it makes sense to consider these deadlines as soft deadlines, because this assumption relaxes the scheduling criteria of application.

Theoretically speaking, each event is linked to the entity that performs the computation and generates response. This entity is often called a task, or a *`computational task`*. Task starts when event arrives, and ends when it generates response. Therefore, computation time *C* is actually an execution time of the computational task. In some cases this task may be mapped directly to the task object in OSEKturbo, though in reality the computational task that processes an event may be an Interrupt Service

Routine of OSEKturbo, or combination of an ISR and OSEKturbo tasks, or combination of OSEKturbo tasks.

Surely, any realistic system processes the number of events simultaneously, and therefore it should provide the response for each event. Each event needs a processor to execute a computational task in order to generate response. As numerous events come independently, the computational tasks compete for processor resource. The Operating System should provide the means of sharing processor between tasks. The main objective of hard real-time system (such as OSEKturbo) is to provide that the processor is shared between computational tasks in best interest of deadlines. In other words, all hard deadlines shall be met, and soft deadlines should be met.

Like most real-time Operating System, OSEKturbo employs two fundamental mechanisms aimed to satisfy the goal of meeting deadlines in an application: the *`fixed-priority scheduling`* and the *`preemptive dispatching`*. Fixed-priority scheduling assigns static priorities to tasks and ISRs, thus allowing the priority ordering demands of the computational tasks for processor according to the "urgency"[1] of the events the computational tasks serve. Preemptive dispatching provides that the task or the ISR that have higher priority than one currently executed on the processor, gets processor immediately when this task or ISR arrives in the application. Therefore, the computational task that serves "urgent" event is not delayed by the computational task that serves less "urgent" event.

As fixed-priority scheduling and preemptive dispatching are cooperating on providing real-time capabilities, the combination of them is usually depicted by the term *`fixed-priority preemptive scheduling`*. In OSEKturbo this kind of scheduling is called *`full-preemptive scheduling`*[2].

1. Urgency is intentionally vague term here. As it will be shown in next sections, in most cases urgency correlates with deadline.

2. While OSEKturbo supports also *`non-preemptive`* and *`mixed-preemptive`* scheduling, the use of them is more subtle, and it is not discussed here for simplification. However, application may benefit from using these types of scheduling - see "Internal Resources," for details.

Objectives of Schedulability Analysis

Main Goal of Schedulability Analysis

Computational tasks, and, hence OSEKturbo tasks and ISRs compete for the processor. In reality tasks and ISRs of higher priority preempt tasks and ISRs of lower priority thus delaying their execution and generation of responses. Therefore, it is not clear if all deadlines of the application are met in all possible conditions of an application execution. Assuming that in reality an application serves dozens and even hundreds events, and consists of dozens of tasks and ISRs with different priorities, the quick glance at an application structure can not provide the answer to the question if all deadlines of application are always met. This is where schedulability analysis came from.

The goal of *Schedulability Analysis* (*SA*) is to check and prove that application is *scheduled*, i.e. that all application deadlines are always met. More specifically, hard deadlines are never missed in application, while soft deadlines are missed occasionally without making harm to an application.

OSEKturbo is designed for the automotive applications, including mission-critical ones. Every and each effort should be made by the application developers to provide that application is reliable. Schedulability analysis is an important part of these efforts, because it is the only mean to verify that application satisfy real-time requirements.

Modeling Approach in Schedulability Analysis

Schedulability analysis employs modeling approach. Schedulability analysis explores the scheduling of the application using the model of application structure and parameters that describe the application timing behavior. This model is called *computational model* in DS-Design Tool.

The timing parameters may differ from real values that application exhibits. For example, the value of computation time may vary depending of amount of calculation needed to generate response. Instead of the real value of computation time the schedulability analysis uses the *worst-case execution time (WCET)*, which is the upper bound of computation time. WCET is the modeling parameter.

Another example of modeling is dependencies between tasks. Simple methods of schedulability analysis may assume that all computational tasks are independent while in reality there are dependencies between tasks in an application (e.g. computational task B starts after computation task A completes).

Modeling approach could provide unrealistic and invalid results if model is improper. There are two basic means that are used to avoid this:

- improving the accuracy and the conformance of the computational model to the real applications,
- and intentionally introducing pessimism in results.

The accuracy and conformance of the computational model are improved by means of exploring and inventing methods of modeling application in terms of timing behavior. DS-Design Tool uses advanced computational model specifically developed for OSEKturbo applications that is described in "Analysis of BCC Applications".

Intentional pessimism means that if analysis shows that application is scheduled, it is scheduled in reality. If the analysis shows that application is not scheduled, it may be scheduled in reality.

One of the example of intentional pessimism is the use of WCET instead of real computation time. Worst-case may never happen in application life-time, so analysis will provide trusted pessimistic results.

Schedulability Analysis and Testing

Schedulability Analysis can not be avoided by means of using testing. Whatever testing approach is taken in the development process, it checks that application *works* in *testing conditions*, even if testing conditions are extreme, or seems to be extreme. Schedulability analysis checks that application is *scheduled* in *any possible conditions*.

NOTE SA checks only timing behavior of the application. For example, SA does check that response to an event is generated in **proper time** (i.e. meets deadline), but it does not check whether the response is **proper** (i.e. the computed response has valid value).

Another important difference between SA and testing is that SA uses *model* of application, while testing usually deals with the *real* application

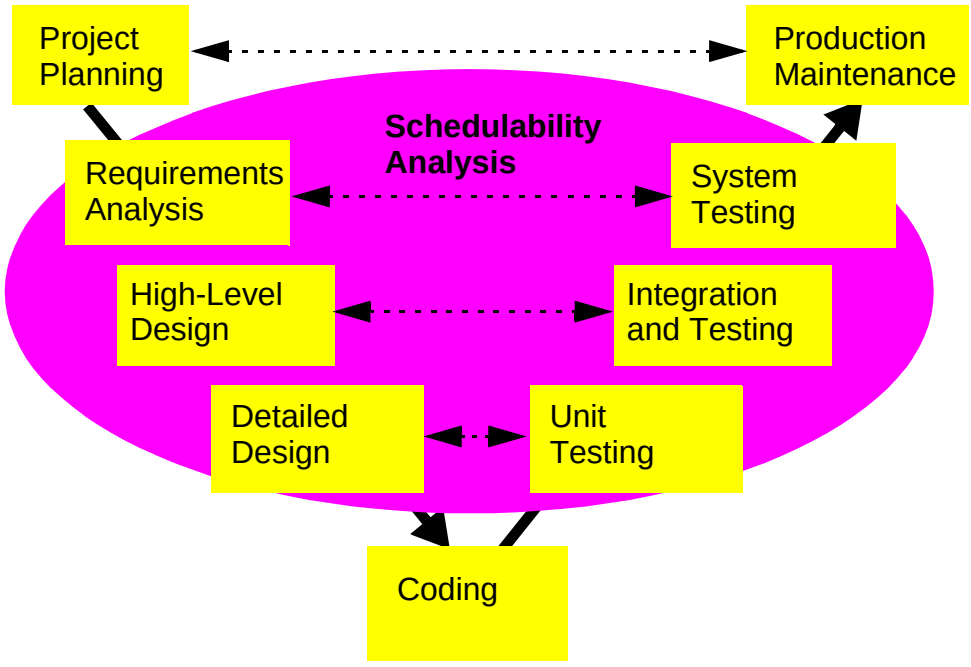
(see **Section Modeling Approach in Schedulability Analysis** for description of the modeling approach).

NOTE Therefore, schedulability analysis is complimentary to testing.

Schedulability Analysis in the Development Process

Schedulability analysis may be used at different phases of the development process. **Figure 3.3** presents stages, where schedulability analysis might be applied in V-shaped development process, often used in automotive software engineering.

Figure 3.3 Schedulability Analysis in V-shaped development process



At earlier stages such as requirements analysis and prototyping SA helps to understand if computing power is enough to handle all events of an application. Structure of the application may be described rather roughly. For example, for each event the computational task is created, and analysis checks if the application is scheduled or not.

At design phase when the application is decomposed into real operating system objects, and computational tasks are mapped onto tasks and ISRs, the schedulability analysis helps to evaluate if design is done properly in terms of scheduling. Different design decisions may be evaluated and refined. Surely, the timing attributes of the application could not be exactly measured, and estimations are used as an approximation.

At detailed design, coding, and unit testing phases the timing of the application is measured, and the numbers are used to re-evaluate the scheduling of application. This is most important use of schedulability analysis. Based on the analysis results application developer makes the changes in code and application structure.

At integration phase schedulability analysis is helpful in checking that integrated application is scheduled.

What Schedulability Analysis Does

Schedulability analysis is first of all an **analysis**. SA helps to evaluate timing behavior of application, to uncover scheduling problems, and to understand what should be done to improve scheduling of application.

Schedulability analysis tool may be compared with logic analyzer often used in software design, because it shows how the software works on the hardware, allowing the developer of application to make necessary changes.

Schedulability analysis is appropriate in statically configured embedded applications such as OSEKturbo applications, because the system objects are known in advance - at an application compile time.

NOTE DS-Design Tool is useful in the development process for verification of timing requirements of application.

What Schedulability Analysis Does Not Do

Schedulability Analysis can not make the application scheduled if it is not scheduled. Analysis can not make better design of application. Analysis can not test the application. Analysis can not produce robust results if inputs to the analysis are vague or inaccurate.

Generally speaking, schedulability analysis can not produce good results if application execution environment is varying. For example, application running on general-purpose operating system are usually **dynamic** applications. It is not known in advance what application would be executed, and therefore it is very difficult to apply to the analysis methods.

The other source of dynamic behavior is code and data caches. The timing of operation may vary significantly depending on cache status, and it is rather difficult to measure timing parameters such as computation time.

NOTE DS-Design Tool does not make applications better, but it does provide the application designer with means and data that help to make better applications.

NOTE DS-Design Tool does not build the schedule of application tasks, but it evaluates the worst-case schedule.

Basic Methods of Schedulability Analysis

Development of schedulability analysis methods for fixed-priority preemptive scheduling started more than three decades ago.

In 1973 C. L. Liu and James W. Layland had shown (see **[7]** in section "References") that application consisting of *fully independent* periodic tasks with relative deadlines equal to periods was optimally scheduled when priorities of tasks were assigned according to *`rate monotonic algorithm`* (i.e. higher priority was assigned to the tasks having shorter period).

The optimally of the rate monotonic algorithm means that if some feasible schedule exists, then the rate monotonic algorithm gives feasible schedule, too. Moreover, the simple test based scheduling was proposed. The application is scheduled if overall utilization of CPU does not exceed *`utilization bound`*.

Utilization bound depends on a number of tasks in application. If application has only one task, the utilization bound is 100%. This is trivial result as only one computational task may fully occupy CPU.

However, as number of tasks grows, the utilization bound decreases, and for infinite number of tasks it has the value ~69%, or ln(2).

The method is called Rate Monotonic Algorithm (RMA), and has the modification called *`Deadline Monotonic Algorithm`* (DMA). DMA considers the tasks with relative deadlines less than task periods, and shows, that in this case optimal scheduling is reached, when the tasks having shorter deadline are assigned with higher priority.

NOTE Rate and Deadline Monotonic Algorithms are in most cases good start for allocation of task priorities in an application.

While this result is quite important milestone from academic point of view, it is not that useful in engineering practices. The reasons for that is the fact that in real-world application events are often dependent, and arrival pattern is not always periodic. In spite that method has been largely extended and developed for realistic applications (for example, see **[5]** and **[6]** in section "References"), it still does not exactly fit engineering cases.

For example, tasks are not always fully independent. Clearly, if one task schedules another task, then they are not started at the same time which is an implicit condition in **[7]** in section "References". Therefore, applying RMA for such application produces *ungrounded* pessimistic result - e.g. it shows that application is not scheduled while in reality it is.

NOTE RMA and DMA demonstrate quite important trade-off in schedulability analysis for applications of real-world: the simplier methods is, the more pessimistic result it produces. This is caused by fact that simplified model does not reflect important particularities of timing behavior of application and operating system.

Advanced Methods of Schedulability Analysis

The need of having analysis methods that are adequate to the real-world application forced researchers to investigate more advanced algorithms. One of them is so-called *`transaction concept`*, described in **[1]** in section "References". This method introduces dependencies between tasks linking them into transaction where each task starts their execution from the static time offset.

The method proposes computational model that fits to many real-world applications. For example, there is *timescale* mechanism in OSEKturbo that links several tasks to timer using time offsets for start point of each task. Clearly, the tasks in timescale can not start independently. Therefore, tasks execution is distributed in time, and the load of CPU is more equalized comparing with case when tasks are independent. Hence, utilization bound is increased, and transaction concept for timescale produces more realistic and less pessimistic result comparing with DMA. Timescale is considered as the transaction, that "drives" tasks.

Application may have several transactions, that are independent from each other. For example, beside timescale application may have *counter* and number of cyclic *alarms* connected to this counter. This will form the second transaction in the application.

Transaction concept has been further developed in **[2]**and **[3]** in section "References".

DS-Design Tool uses an extension of transaction concept that employs *`subtransactions approach`*. The breakdown of transaction into subtransactions allows more accurate and realistic description of application timing behavior thus decreasing pessimism of analysis even further (comparing with transaction concept). Continuing timescale example, each *timestep* in timescale is considered as subtransaction of timescale transaction. Subtransaction may be a composition of several tasks, or even *task sections*. DS-Design Tool considers scheduling of subtransactions, thus providing reliable analysis of OSEKturbo applications of *Basic Conformance Classes (BCC).*

The use of subtransaction approach in DS-Design Tool allows to provide schedulability analysis for applications of OSEKturbo *Extended Conformance Classes (ECC).* This is a unique feature of the tool.

As OSEKturbo supports *internal resources*, DS-Design Tool uses the extension of methods of schedulability analysis originally developed for tasks with *`preemption threshold`* (see **[10]** in section "References").

4

Tool Architecture

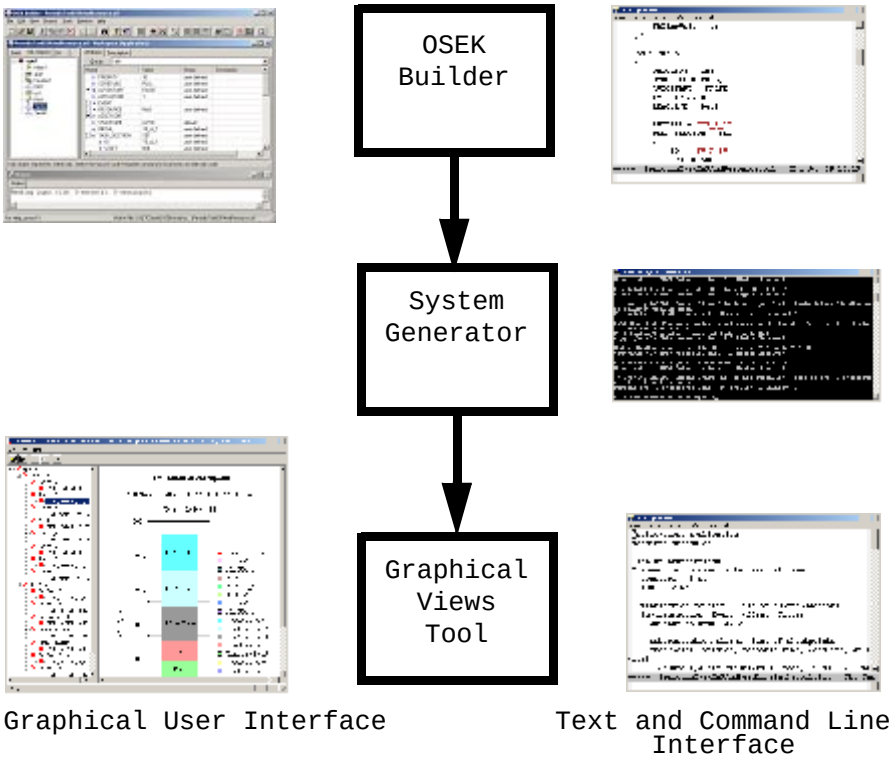
This chapter consists of the following sections:

- DS-Design Tool Components
- OSEK Builder
- OIL File
- Schedulability Analysis
- Graphical Views

DS-Design Tool Components

DS-Design Tool components are presented on the following picture:

Figure 4.1 DS-Design Tool components



Basically, DS-Design Tool is integrated into OSEKturbo products, and can be used as graphical interface tool, or as text and command line tool.

In a graphical interface mode, an application designer uses OSEK Builder to define application timing behavior. All timing information is stored in OIL files, therefore OSEK Builder provides all needed functionality to enter and edit timing parameters. OSEK Builder launches System Generator to perform schedulability analysis of an application. The System Generator produces *`graphical views file`*, that contains results of the schedulability analysis[1]. This file is visualized by means of using *`Graphical Views Tool`* (GraViTool), that is also launched from OSEK Builder. GraViTool presents the scheduling information in a series of charts.

1. Graphical Views File format is the XML format file.

In a text and command line mode, the application designer uses conventional text editor to edit OIL file and describe application timing behavior there. Then the application designer starts System Generator from command prompt. System Generator produces same graphical views file as it is described above. Then the application designer starts GraViTool and directs the tool to generate *`text views`* of the application. The text views contain basically same information as graphical charts of GraViTool, but they are plain text files, and can be viewed by conventional text editor.

Both graphical interface mode and text and command line mode can be used in a mixed manner.

On the picture the graphical interface mode is shown on the left side, while the text and the command line mode are shown on the right side.

OSEK Builder

DS-Design Tool uses conventional OSEK Builder that has add-on for schedulability analysis. In particular, there are buttons on toolbar and items in Project menu that are used to perform schedulability analysis and to launch GraViTool.

Also DS-Design Tool specific features are supported by means of using *`SA Implementation`*. Therefore, SA tabs are introduced in Workspace, Output, etc.

OIL File

DS-Design Tool specific parameters of application are fully described in standard OIL file. There is no deviations from OIL standard (see **[8]** in section "References,"). All DS-Design Tool specific parameters are described in sections Analysis of BCC Applications and Analysis of ECC Applications, and summarized in OIL Language Quick Reference.

Schedulability Analysis

Schedulability analysis is performed by special dynamic link library of System Generator. The specific options are used to direct System Generator to perform schedulability analysis instead of generating

configuration code. These options are used automatically by OSEK Builder, or can be used explicitly as described in Batch Mode of Analysis.

Graphical Views

The graphical views component is a standalone utility called GraViTool that normally is stared from OSEK Builder. The GraViTool functionality and charts are described in sections Analysis of BCC Applications and Analysis of ECC Applications. The use of text views is described in Batch Mode of Analysis.

5

Getting Started

This chapter consists of the following sections:

- Application Sample
- Defining Application Timing Parameters
- Starting Analysis
- Exploring Analysis Results
- Changing the application timing behavior

Application Sample

The sample application has three independent periodic tasks *Task2ms*, *Task4ms*, and *Task8ms*. The periods of tasks are 2 milliseconds (ms), 4 ms, and 8 ms respectively.

Each task has computation time 1 ms, and each task should complete the computation before its deadline. The tasks deadlines are equal to the periods of tasks.

As soon as each task completes computation, it terminates itself.

There is a counter in an application that generates tick each millisecond. The name of the counter is *Counter1ms*.

The periodic activations of tasks are made by means of using three alarms - one per task. All alarms are started automatically during Operating System startup. For the sake of convenience the alarms are named *Alarm2ms*, *Alarm4ms*, and *Alarm8ms* for respective tasks *Task2ms*, *Task4ms*, and *Task8ms*.

The application uses full preemptive scheduling policy. The priorities of the tasks are assigned according to Rate Monotonic Algorithm described in "Basic Methods of Schedulability Analysis" on page 23. Therefore, *Task2ms* has priority 10 (high), *Task4ms* has priority 9 (middle), and *Task8ms* has priority 8 (low).

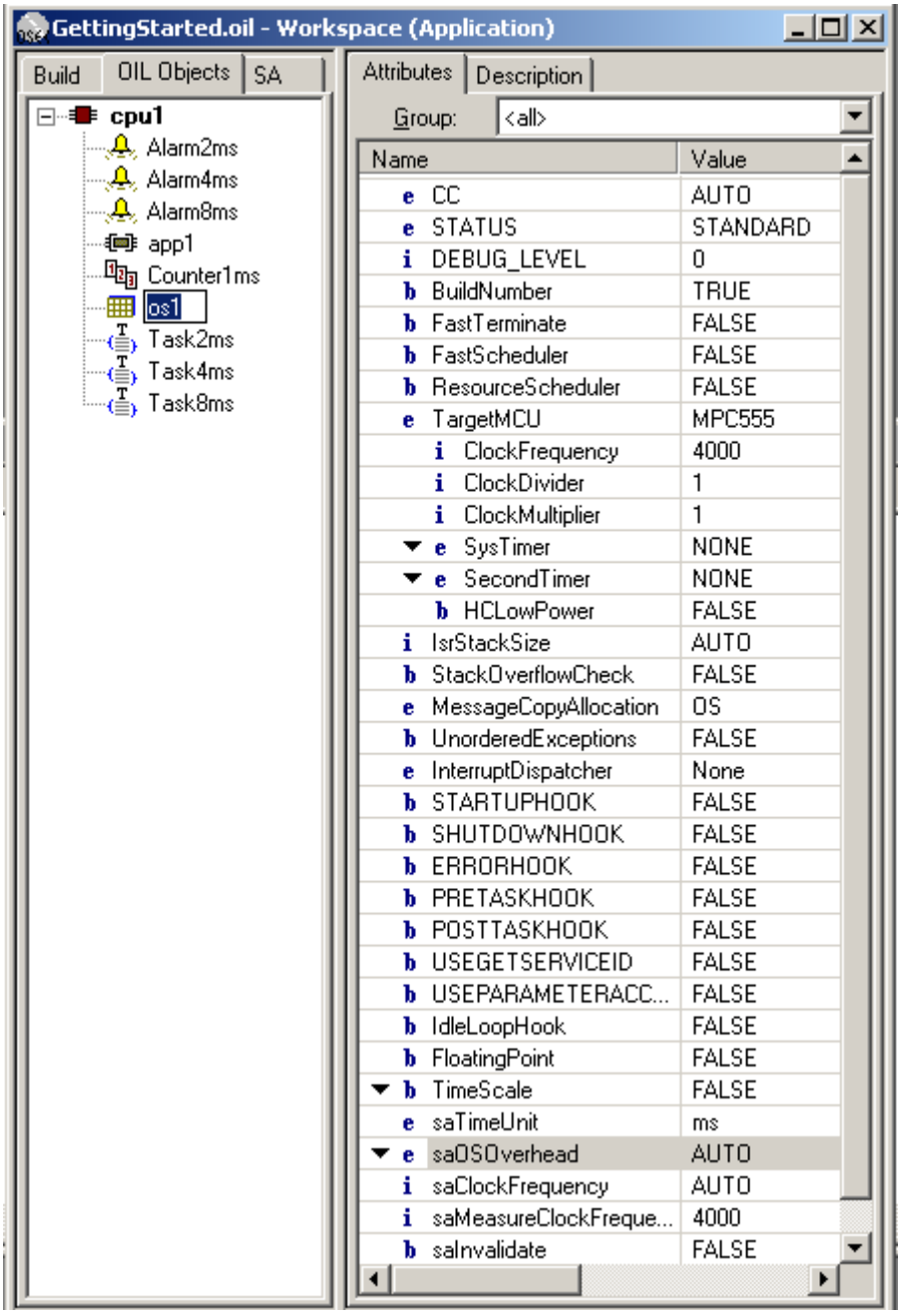
The question that should be answered is “are all deadlines met in this application?”.

Defining Application Timing Parameters

The timing parameters of application are defined by means of using OSEK Builder. The application file has the name *GettingStarted.oil*[1].

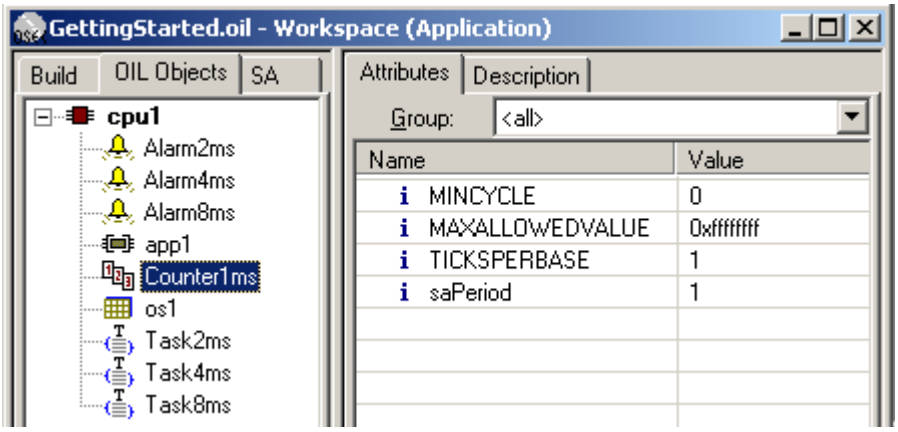
1. The text of the file is reproduced in the end of this section

First all system objects should be created and parameters of OS object *os1* should be defined:

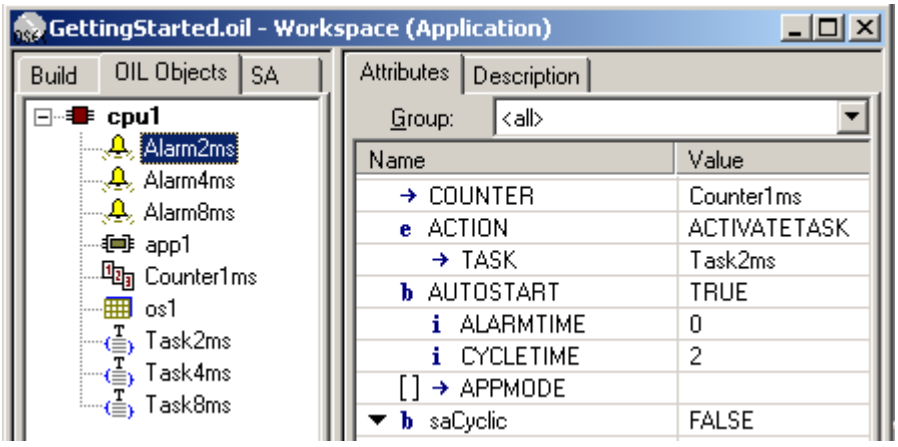


The OSEKturbo OS/MPC5xx version is used in examples, but but the actual view may vary according to the OSEKturbo version installed.

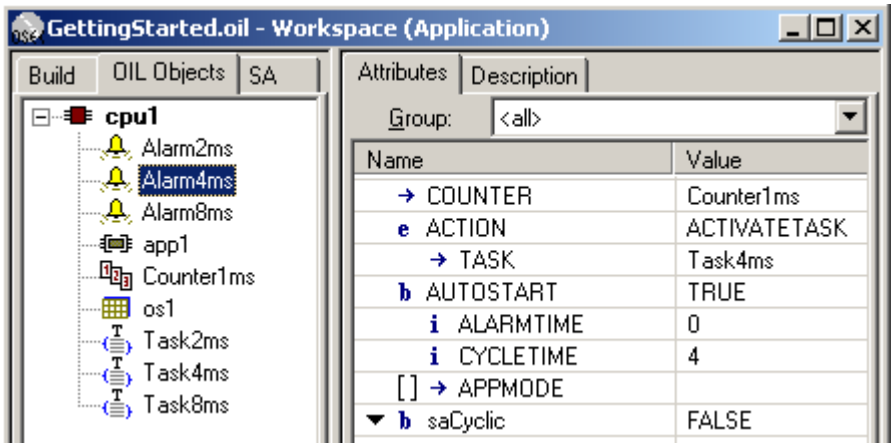
Next, counter *Counter1ms* should be created:



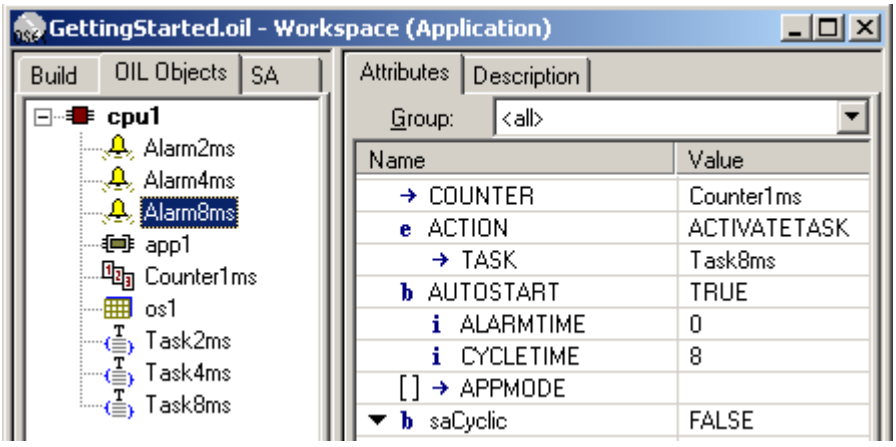
Then, three alarms should be created. *Alarm2ms* has the following values of parameters:



Alarm4ms has the following values of parameters:



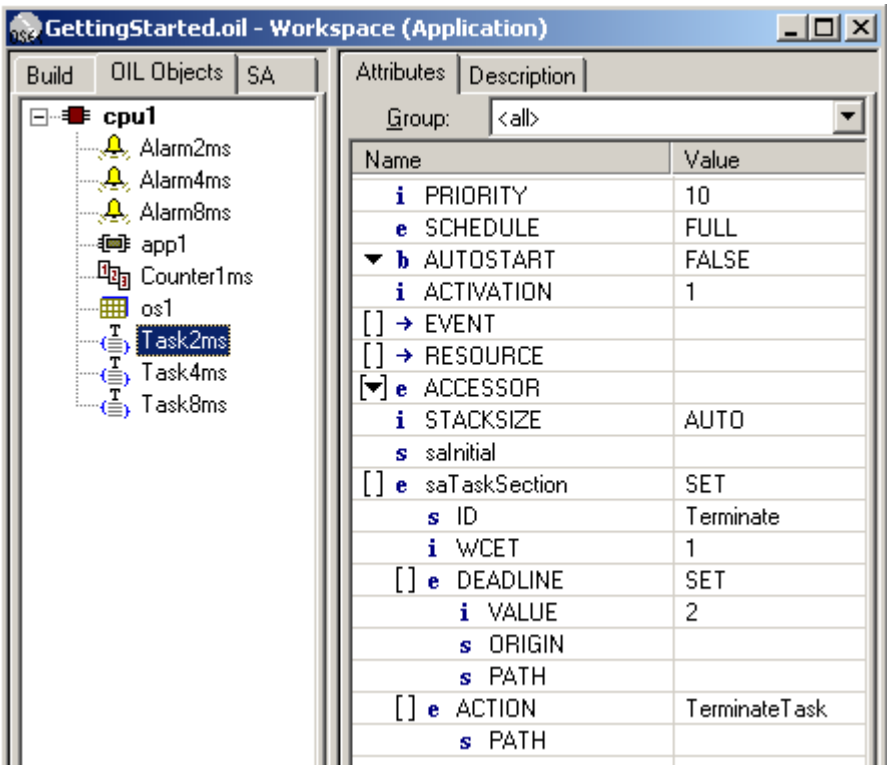
Alarm8ms has the following values of parameters:



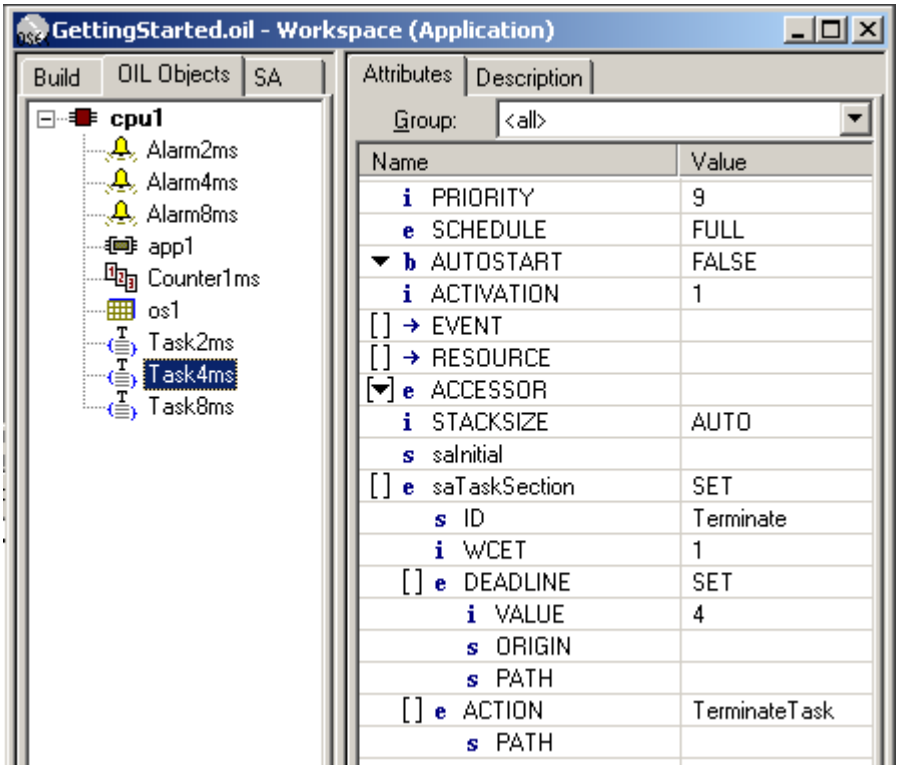
Then, tasks should be created. Each task has one task section that covers whole body of the task, and therefore task section[1] has worst-case computation time 1 ms. The name of the section is *Terminate*. For each task the deadline of the section is different.

[1] The task sections and the ISR sections are used by DS-Design Tool for schedulability analysis as it is explained in next sections of this document.

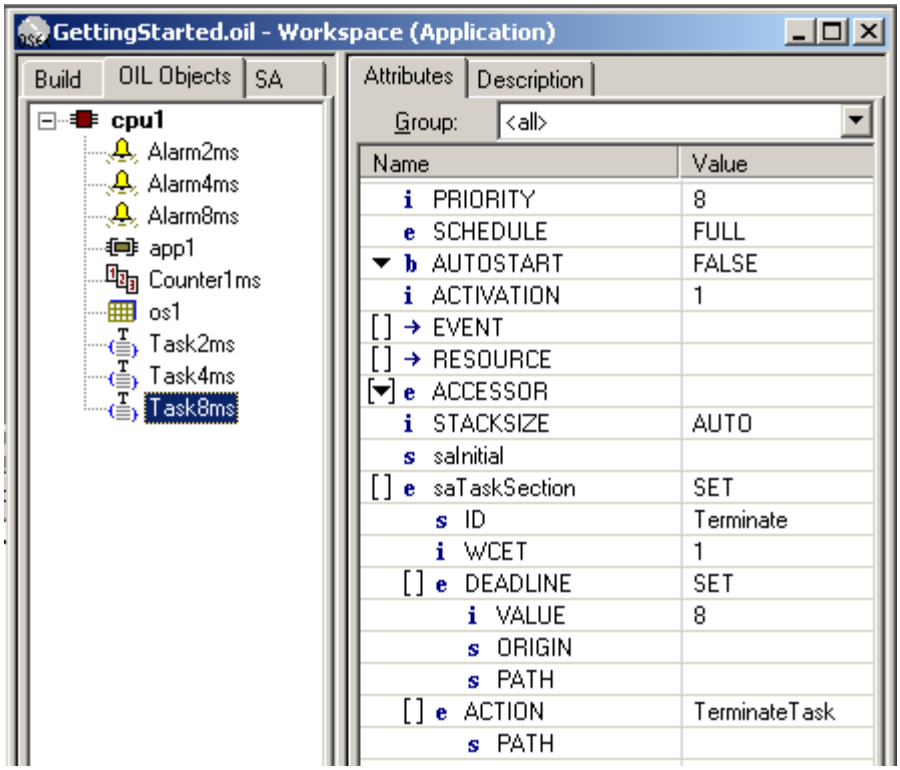
Task2ms has the following values of parameters:



Task4ms has the following values of parameters:



Task8ms has the following values of parameters:



Starting Analysis

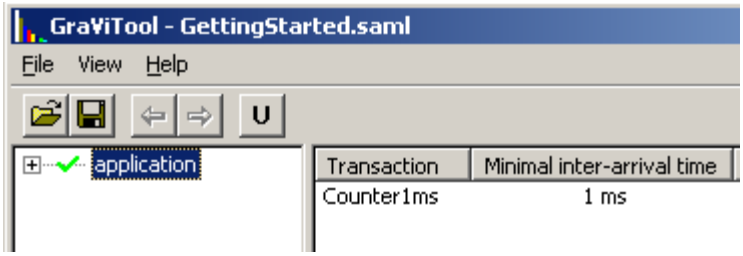
Now when all parameters are entered, DS-Design Tool should be instructed to start the analysis. This could be done either via menu item Project|Analyze, or by means of clicking SA tool on the toolbar:

The OSEK Builder launches System Generator, which performs analysis.

Exploring Analysis Results

When analysis is done, GraViTool should be started to visualize the results of schedulability analysis of an application. This could be done either via menu item Project|View, or by means of clicking View tool on the toolbar:

GraViTool opens the main window:

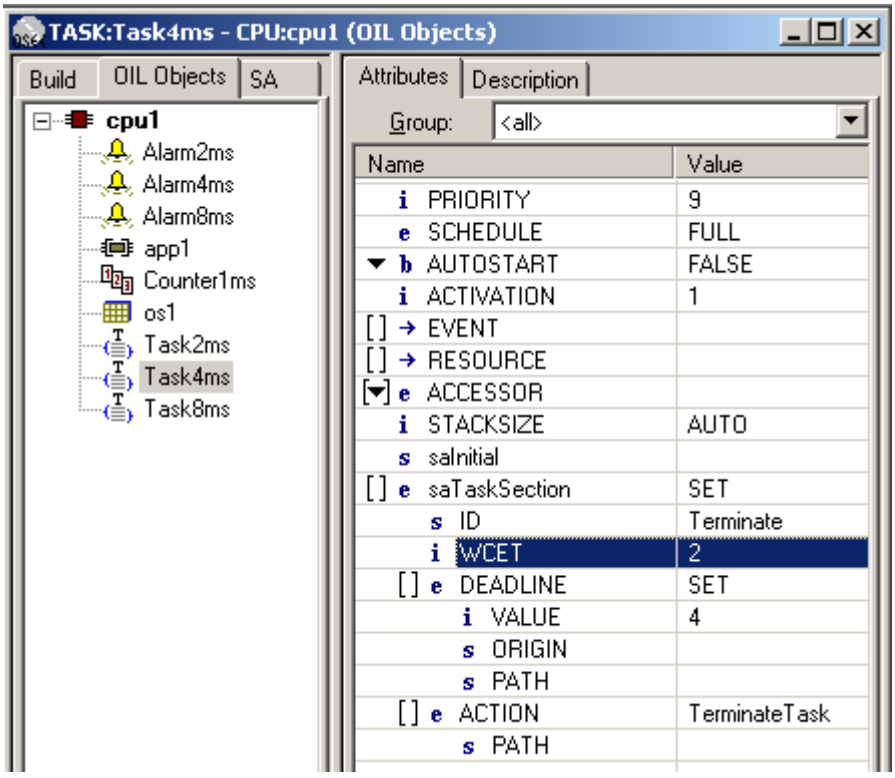


The green vote mark on the application tree prompts that application is *scheduled*, because the deadlines of all three tasks are met.

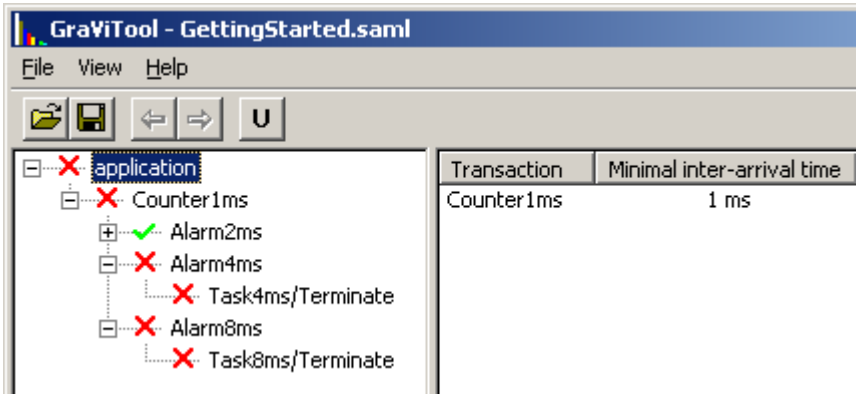
Changing the application timing behavior

Now, suppose that *Task4ms* requires 2 ms of computation time to complete its work. Will the application be scheduled then?

The only change is the value of *WCET* in *Task4ms*:



After performing analysis the GraViTool shows that application is not scheduled indeed:

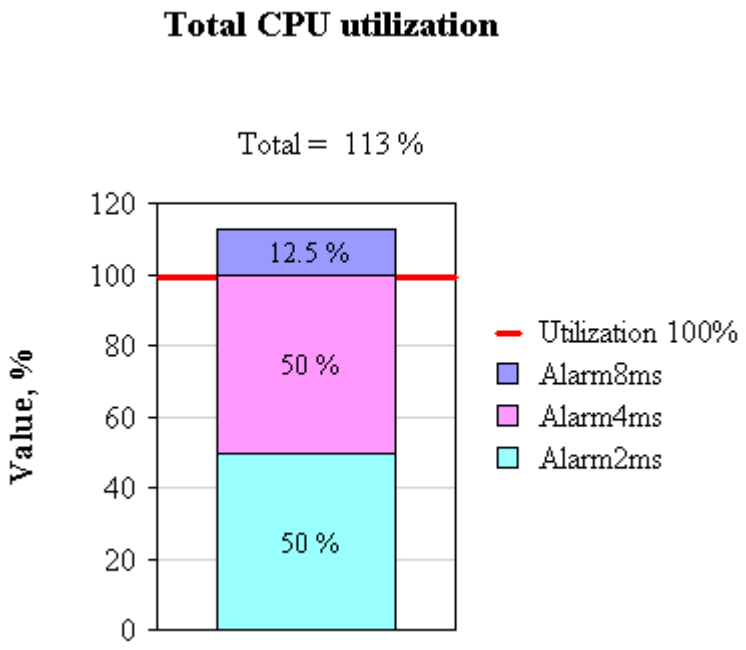


The red scratch mark on the application tree branches prompt that deadlines of *Task4ms* and *Task8ms* are not met.

Clicking on utilization tool in GraViTool toolbar

shows the reason of that - the over utilization of CPU:

Figure 5.1 Total CPU utilization view



GettingStarted.oil file content

Here is complete OIL file that can be opened in OSEK Builder. DS-Design Tool specific parameter are in **bold**:

```
// GettingStarted.oil

OIL_VERSION = "2.3";
#include "C:\metrowerks\osek\ostmpc\bin\ost22mpc.oil"

CPU cpu1
{
    APPMODE app1;
    OS os1
    {
      TargetMCU = MPC555;
      STATUS = STANDARD;
      ResourceScheduler = FALSE;
      STARTUPHOOK = FALSE;SHUTDOWNHOOK = FALSE;
      PRETASKHOOK = FALSE;POSTTASKHOOK = FALSE;ERRORHOOK = FALSE;
      USEGETSERVICEID = FALSE; USEPARAMETERACCESS = FALSE;

      saTimeUnit = ms;
      saMeasureClockFrequency = 4000;
    };

    TASK Task2ms
    {
      PRIORITY = 10;
      SCHEDULE = FULL;
      AUTOSTART = FALSE;
      ACTIVATION = 1;

      saTaskSection = SET { ID = "Terminate";
          WCET = 1;
          DEADLINE = SET { VALUE = 2; };
          ACTION = TerminateTask;
      };
    };
    TASK Task4ms
    {
      PRIORITY = 9;
      SCHEDULE = FULL;
      AUTOSTART = FALSE;
      ACTIVATION = 1;

      saTaskSection = SET { ID = "Terminate";
          WCET = 1;
          DEADLINE = SET { VALUE = 4; };
          ACTION = TerminateTask;
      };
```

```
    };
    TASK Task8ms
    {
      PRIORITY = 8;
      SCHEDULE = FULL;
      AUTOSTART = FALSE;
      ACTIVATION = 1;

      saTaskSection = SET { ID = "Terminate";
          WCET = 1;
          DEADLINE = SET { VALUE = 8; };
          ACTION = TerminateTask;
      };
    };

    COUNTER Counter1ms
    {
      MAXALLOWEDVALUE = 0xffffffff; MINCYCLE = 0; TICKSPERBASE = 1;
      saPeriod = 1;
    };

    ALARM Alarm2ms
    {
      COUNTER = Counter1ms;
      ACTION = ACTIVATETASK { TASK = Task2ms; };
      AUTOSTART = TRUE { ALARMTIME = 0; CYCLETIME = 2; };
    };
    ALARM Alarm4ms
    {
      COUNTER = Counter1ms;
      ACTION = ACTIVATETASK { TASK = Task4ms; };
      AUTOSTART = TRUE { ALARMTIME = 0; CYCLETIME = 4; };
    };
    ALARM Alarm8ms
    {
      COUNTER = Counter1ms;
      ACTION = ACTIVATETASK { TASK = Task8ms; };
      AUTOSTART = TRUE { ALARMTIME = 0; CYCLETIME = 8; };
    };
};
```

6

Analysis of BCC Applications

The section describes the computational model of OSEKturbo applications that are conformed to Basic Conformance Classes. This computational model is used by DS-Design Tool to perform schedulability analysis of the BCC applications.

This chapter consists of the following sections:

- General
- Transactions
- Subtransactions
- Tasks and ISRs Sections
- Timepoints and Checkpoints
- Execution Paths
- Applying Computational Model
- Tuning an Application
- Limitations for the Application Structure
- Dictionary of DS-Design Tool

NOTE DS-Design Tool is capable to analyze applications of Extended Conformance Classes. Most from this section is applicable to the analysis of the basic tasks of ECC applications. Particularities of the extended tasks analysis are covered in "Analysis of ECC Applications".

The section describes the computational model itself and explains how an application is mapped to the model.

Subsection **General** introduces computational model.

Subsections **Transactions** and **Subtransactions** present the hierarchy of the model's main building blocks and explaining how transaction and subtransaction are used for the model the application timing behavior.

Subsection **Tasks and ISRs Sections** explains the breakdown of tasks and ISRs into execution sections, which are the minimal entities used by DS-Design Tool for schedulability analysis.

Subsection **Timepoints and Checkpoints** describes the deadline watching means of DS-Design Tool.

Subsection **Execution Paths** explains how the model may be used to analyze complex tasks and ISRs that have branches.

Subsection **Applying Computational Model** explains on examples how the model should be applied to the real-world OSEKturbo applications. Also subsection presents detailed description of the results that are generated by DS-Design Tool and explains how these results could be used to explore timing behavior of application to see if it is proper or not. In later case the subsection **Tuning an Application** contains description of means that are used to improve timing behavior, and explains how these means should be used.

As any model the computational model of DS-Design Tool has limitations that are described in **Limitations for the Application Structure**.

Dictionary of DS-Design Tool is presented in **Dictionary of DS-Design Tool**.

General

DS-Design Tool needs information about timing behavior of application in order to perform schedulability analysis. This information is provided via OIL language parameters, which are described in OIL configuration file or in OSEK Builder dialogs.

DS-Design Tool uses the model of application because the schedulability analysis deals with the semantics of an application. DS-Design Tool needs to know what events are served by the applications, which tasks and interrupt service routines are used to handle the events, and which deadlines are set by the application designer for the responses.

NOTE DS-Design Tool may be used with DS-V module[1] that allows automated gathering of subset of an application timing data. For example, computational times of tasks and ISRs can be extracted from trace of the application provided by DS-V, and can be used an estimate of worst-case execution times of corresponding entities.

The computational model is composed from the following entities:

- Transactions
- Subtransactions
- Tasks and ISRs sections
- Timepoints and Checkpoints

At functional level, computational model defines the following data:

- external events of an application with the defined timing data (e.g. period of system timer),
- tasks and ISRs that handle the events,
- deadlines for the responses that should be generated by the application as a result of events processing,
- particularities of execution of tasks and ISRs - e.g. processing levels allocated to them,
- overhead of executing operating system activities - e.g. timing of OS system services.

NOTE In order to simplify explanation, operating system overhead is ignored in this section. However, DS-Design Tool takes into consideration the OS overhead - see "Dealing with Operating System overhead".

Basic Conformance Classes of OSEKturbo supports only basic tasks, that do not have waiting state. Basic tasks are executed from the beginning till self-termination though they may be preempted by tasks of higher priority, and may be interrupted by ISRs.

1. The DS-V tool is available as a separate product.

Transactions

Transactions Overview

The execution entities of OSEKturbo Operating System are tasks and interrupt service routines. The tasks are fully managed by Operating System. The interrupt service routines are managed in a limited sense by Operating System, because scheduling of ISRs depends on a hardware platform.

Tasks and ISRs are the only entities of OSEKturbo application capable to generate responses to the events, and thus providing real-time processing. OSEKturbo is designed to support *event driven* application. That means that an event “drives” the execution of the entity or entities that generate response for the event.

NOTE In most cases the events come from the hardware, which is connected to sensors, timers, network, etc. These events are called environmental, or external events, because they are generated in an outer world.

External events are *independent* from each other. Indeed, timer ticks and reception of network messages from CAN bus are the completely independent external events. Therefore, interrupt requests from timer and from CAN controller come independently.

To analyses the handling of external events, DS-Design Tool uses *`transactions`*. Transaction is a set of application execution entities which are executed by the operating system to respond to an external event.

For example, OSEKturbo system timer tick is an external event. The application responds to this event by means of using alarm attached to this timer, and by activating the task when system timer ticks reach the predefined number. The task provides response to the system timer tick event, and terminates. Therefore, the alarm’s task is an application execution entity that responds to the external event. Thus, the transaction for system timer consists of the alarm attached to the system timer, and of the task that is linked to the alarm.

NOTE The application may have several alarms attached to the single counter (timer), which are described and analyzed in DS-Design Tool by means of using subtransactions (see **Section Subtransactions**).

Another example is an ISRs that handles the reception of network messages from CAN bus and activates the task that processes the messages to generate control signals for an actuator (e.g. window lift motor). The event here is a reception of the data frame from CAN bus. The execution entities that respond to this event are the ISR and the task. Thus, the transaction for CAN message reception consists of the ISR and the task.

NOTE Independent release (the occurrence) of events is a main reason why DS-Design Tool uses transactions. As a matter of fact, independency of external events makes the scheduling of application difficult, because processor needs to compute responses in a highly competitive manner, if events occur simultaneously. Actual response times will be the largest for events that are processed at low priorities.

NOTE In an event driving computing concept the transactions are “guiders” of events, because transaction feeds the event into an application. In other words, transaction links an event (more precisely, it links source of the event) and the execution entities that compute the response to the event.

Application Transactions

DS-Design Tool creates transactions automatically using OSEKturbo application configuration information. Depending on the application configuration, the following transactions are created:

- *system timer* transaction
- *second timer* transaction
- *TimeScale* transaction[1]
- transaction for each counter
- transaction for each user-defined *ISR*

Periodic and Sporadic Transactions

In real-time applications events have different *arrival pattern,* which defines events' occurrences are distribution in time. DS-Design Tool provides analysis of *periodic events* and *sporadic events*.

Periodic events are released periodically. There is the constant time interval between consequent releases of periodic event DS-Design Tool creates *periodic transactions* for periodic events. OSEKturbo system timer and timescale are the examples of periodic events, and therefore for system timer and timescale periodic transactions will be created.

DS-Design Tool uses timing parameters of OSEKturbo objects to learn the value of period for the periodic transactions from configuration information in OIL file of an application. OSEKturbo system and second timers and timescale contain the value of period. OSEKturbo counter does not contain information about period of the counter, so DS-Design Tool adds the period parameter to the description of *COUNTER*.

NOTE DS-Design Tool is oriented towards schedulability analysis of periodic computations, because it is the natural paradigm for hard real-time applications.

Unlike periodic events, there is no constant time interval between consequent releases of sporadic events. However, there is *minimal interarrival time* (interval) between consequent releases. That means the sporadic events are released any time, but they can not be released arbitrary often.

Sporadic events are handled by OSEKturbo interrupt service routines, and DS-Design Tool creates the *sporadic transaction* for each ISR in the application.

Interrupt service routines are released not immediately when external event arrives. Hardware introduces the latency, which is called *release jitter* in the DS-Design Tool. The value of the jitter depends on hardware particularities.

[1] TimeScale transaction and system timer transaction are mutually exclusive, because in the OSEKturboTimeScale is attached to system timer.

DS-Design Tool adds the minimal interarrival time and the jitter parameters to the description of an ISR in OIL file.

Subtransactions

Subtransactions Overview

Transaction may link more than one computing entities to the same source of event. Each computing entity is called *`subtransaction`* of the transaction. Subtransactions respond to the event released by the source of event the transaction is linked to. Subtransaction is activated when the event arrives.

NOTE Subtransactions do not exist independently. Each subtransaction belongs to one exact transaction.

NOTE Definition of transactions and subtransactions is a little ambiguous, when there is only one subtransaction in the transaction, because in this case source of event and event itself are essentially the same. This is often the case for interrupt service routines. However, DS-Design Tool always creates the subtransaction for such specific transaction. Therefore, the transaction which consists of just one subtransaction is common in DS-Design Tool.

The clear example of subtransactions is OSEKturbo TimeScale. TimeScale enables periodic activations of tasks in accordance with statically defined schedule. TimeScale is attached to system counter, and it consists of steps. Each step activates one or several tasks, if the timer value reaches the value statically defined on the step. Therefore, while the system timer advances, it reaches steps on TimeScale, and activates the task(s) of the step. The source of event is a system timer, while events themselves are timer ticks.

The most important feature of the subtransactions of the transaction is that the subtransactions within transaction **may** be dependent from each other.

Indeed, on TimeScale two steps can not be reached simultaneously, because there is a time interval between them. Therefore, the tasks of two steps in a TimeScale can not be activated independently.

NOTE Dependent activations (the releases) of subtransactions in transaction is a main reason why DS-Design Tool employs subtransactions. As a matter of fact, dependencies between subtransactions 1) reflect the reality of events arrival in the application, and 2) relax the scheduling criteria of the application. Demands for processor from the subtransactions are distributed in time, so it is easier for processor to execute the subtransactions' tasks and ISRs.

Several alarms attached to one counter (e.g. the second timer of OSEKturbo) are often used in applications. Therefore, counter and timer transactions typically consist of several subtransactions.

NOTE Subtransactions in the counter and timer transaction are considered as independent ones. Unlike timescale, there is no constant time offset between alarms activations attached to the counter or timer. Therefore, DS-Design Tool considers these subtransactions as independent entities.

Applications Subtransactions

DS-Design Tool creates the subtransactions automatically using OSEKturbo application configuration information about relations between application objects. For example, information about attachments of tasks to counters (timers) is defined in *ALARM* objects. However, OSEKturbo OIL parameters do not contain enough information about connections between ISR and tasks. Therefore, DS-Design Tool adds specific attributes to *ISR* and *TASK* objects that define the subtransactions (see ***Section Tasks and ISRs Sections***).

Depending on the application configuration, the following transactions typically has several subtransactions:

- *system timer* transaction (several alarms attached)
- *second timer* transaction (several alarms attached)
- *TimeScale* transaction (several steps)
- counter transactions (several alarms attached)

The following transactions by default consist of a single subtransaction:

- ISR transaction

OSEKturbo alarms are heavily used in applications - especially when periodic events are processed. The application uses operating system services *SetRelAlarm* and *SetAbsAlarm* to start the alarm in either single mode or cyclic mode. This information is also necessary for DS-Design Tool. Therefore, for *ALARM* object DS-Design Tool adds OIL parameters that define the single mode and cyclic mode values of the alarm.

NOTE Autostart alarms are handled by the subtransactions of the transaction of the counter to which alarm is attached. Single mode and cyclic mode values of the autostart alarms are defined in AUTOSTART parameter of ALARM, and therefore there is no need to specify same parameters separately.

Tasks and ISRs Sections

Tasks and ISRs Sections Overview

Transactions and subtransactions are only a framework for the application execution entities, namely tasks and ISRs, that perform computation needed to respond to the event. Therefore, DS-Design Tool uses timing information about tasks and ISRs by means of partitioning them into *`task sections`* and *`ISR sections`* correspondingly. Task section and ISR section are the minimal execution entities that are the subjects of analysis[1].

Briefly, splitting the task and ISRs into sections provides the following advantages:

- subtransactions may be described as a sequence of task and ISR sections thus reflecting the processing algorithms in the application code;
- multiply deadlines may be set to watch the processing of events (each task section and ISR section may have deadline);
- schedulability analysis becomes more accurate as DS-Design Tool takes into consideration timing behavior of each section.

[1] To simplify the explanation, term *section* will be used when there is no difference between task section and ISR section.

Each task section and ISR section defines the following parameters:

- worst-case execution time of the section,
- deadline[1] for the completion of the section,
- action[2] that is performed **at the end** of the section execution.

NOTE Each Task section and ISR section has an unique symbolic name *ID* within the scope of the task or ISR the section belongs to.

Task of Single Task Section

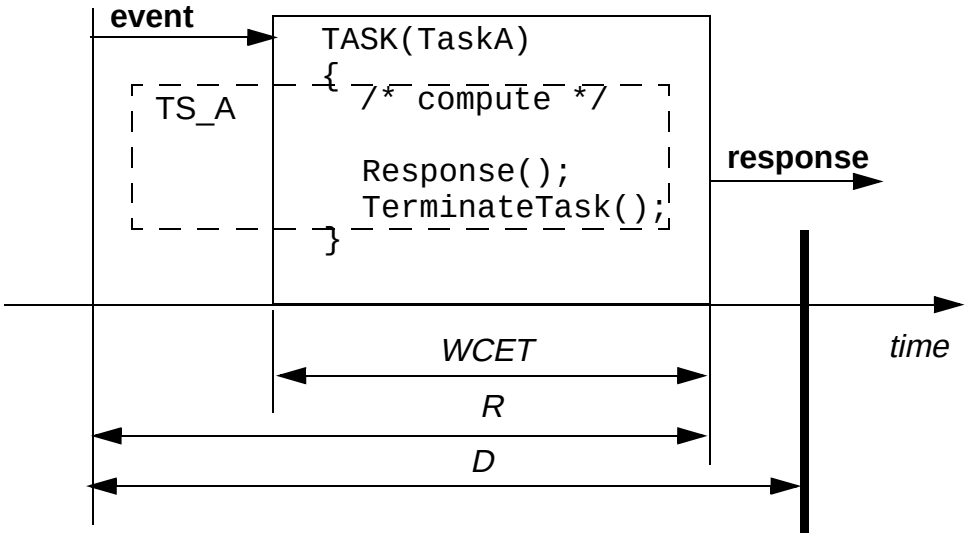
In a simple case, a task starts when the event arrives (e.g. task is activated by alarm), computes the response, generates the response, and terminates itself. As it is shown on **Figure 6.1**, this task consists of a single task section TS_A. The section starts from the beginning of the task body, and completes by the call of system service *TerminateTask*. The call of this system service is an *`action`* of this task section. The timing parameter of the task section are *WCET* (upper bound of computational time needed to compute the section code and generate the response), and deadline value *D*. DS-Design Tool computes the actual response time *R* of this task section, and checks if it meets *`deadline`* of the section.

NOTE Deadline and response time of the subtransaction are relative to the event arrival.

1. There may be more than one deadline set for the section as it is described in **Execution Paths**.

2. There may be more than one action defined for the section as it is described in **Execution Paths** and in cases when GoTo actions are used in the section.

Figure 6.1 Simple task section

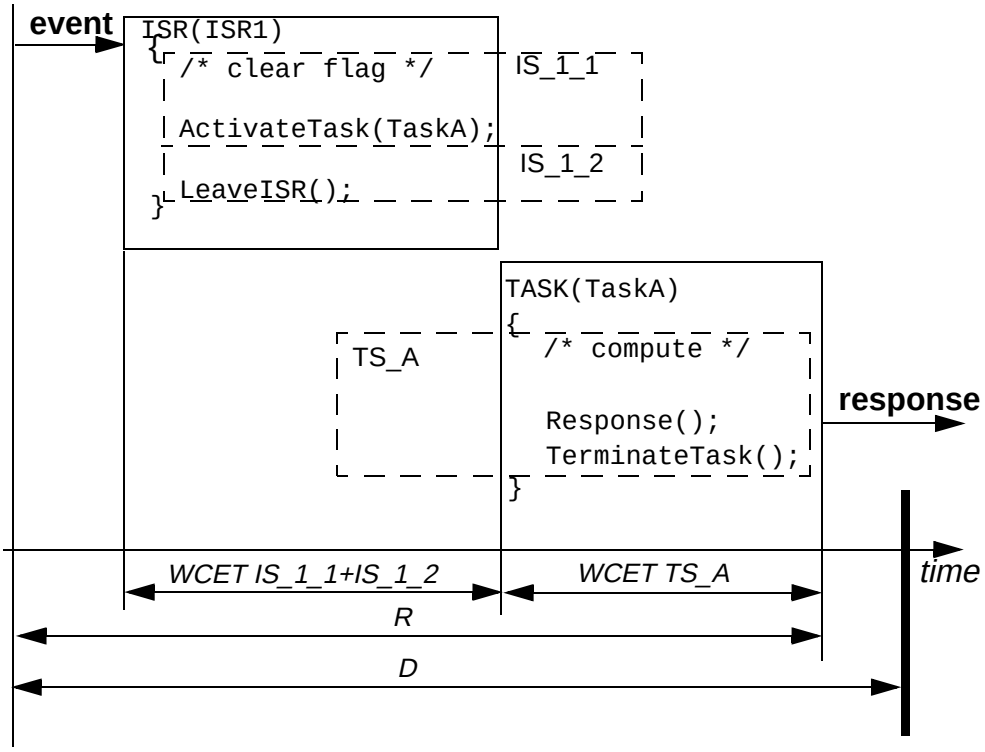


NOTE Each termination of a task or an ISR should be included into task section or ISR section. The task section should have an action TerminateTask (or ChainTask). The ISR section should have an action LeaveISR. DS-Design Tool needs these section to learn when the task or ISR completes the execution.

Sections and Execution Scenario

Though the deadline processing presented on **Figure 6.1** it is applicable in many cases, more complex event processing is required. As an example, **Figure 6.2** presents an ISR subtransaction that responds to an event by means of using an ISR and task. Interrupt service routine ISR1 gets activated when interrupt request comes from the hardware (e.g. CAN module). ISR1 clears the hardware interrupt flag, and activates the task TaskA, that completes the computation needed to generate response, generates response, and terminates itself.

Figure 6.2 Processing an event by an ISR and task



As **Figure 6.2** shows, the ISR1 consists of two ISR sections - IS_1_1, and IS_1_2. Section IS_1_1 ends with action *ActivateTask* specifying the name of the task TaskA that should be activated to complete the processing of the event. After completing execution of section IS_1_1 the ISR1 should continue with section IS_1_2, which ends with action *LeaveISR*. Therefore, section IS_1_1 specifies that *`next`* section is section IS_1_2.

NOTE If the task or the ISR consists of more than one section, the section that should be executed first, is defined by INITIAL parameter of the TASK or the ISR object.

In OSEKturbo the interrupt service routines have higher priority than tasks. Therefore, the following sequence of ISR sections and task section is executed by processor to respond to the event: IS_1_1, IS_1_2, TS_A. This sequence is called *`execution scenario`* of the subtransaction. DS-Design Tool uses the execution scenario to evaluate the response time

taken into considerations the total worst-case execution time of all sections that the subtransaction comprises, and how the sections of the subtransaction compete with other tasks and ISRs.

Execution scenario is derived from an *execution graph* of the subtransaction. Execution graph consists of the set of possible sequences of task sections and ISR sections, that **may** be executed when the subtransaction responds to the event. Execution graph reflects the branches in application algorithms. If there is no branches in the code of tasks and ISRs that the subtransaction comprises, then there is only one execution scenario of the execution graph.

DS-Design Tool uses the links between section, defined in NEXT parameter of section to build execution graph within the task or an ISR. DS-Design Tool uses the references to the system objects, defined in ACTION parameter of section to build the execution graph across tasks and ISRs. DS-Design Tool builds execution scenarios of the execution graph using the *GoTo* actions[1] and *PATH* parameter of the task sections and ISR sections[2].

Sections and Rescheduling points

Another example of an execution scenario is presented on **Figure 6.3**. The subtransaction consists of two tasks - TaskA and TaskB. Subtransaction starts when TaskA is activated because an event arrives (e.g. it is the step of the timescale). TaskA does computation, and activates TaskB that should complete the processing of an event and generate response. Meanwhile TaskA continues execution and terminates itself.

1. *GoTo* action is described in subsection ***Sections and Conditional Operators***

2. *PATH* parameter is described in subsection ***Using Paths for Dependent Conditional Operators***

Figure 6.3 Processing an event by two tasks

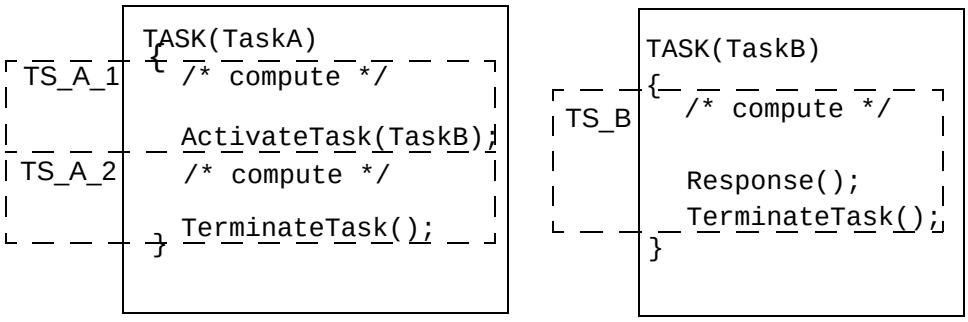


Figure 6.4 Processing an event by two tasks (execution scenario 1)

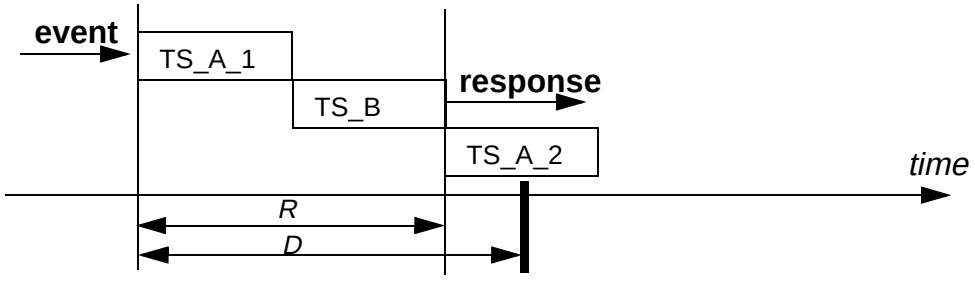
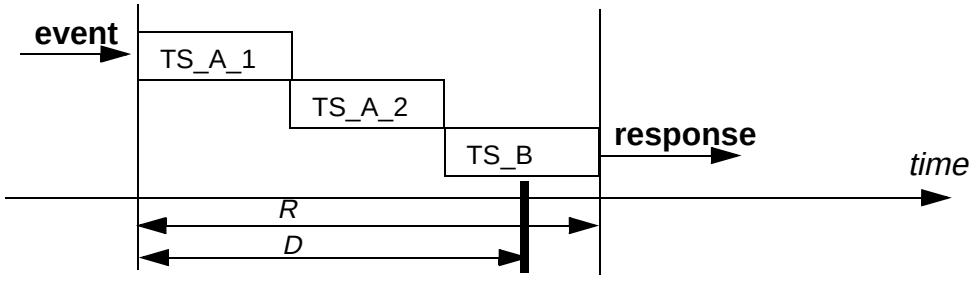


Figure 6.5 Processing an event by two tasks (execution scenario 2)



There are two task sections in TaskA: TS_A_1 and TS_A_2. Section TS_A_1 ends with an action *ActivateTask* specifying TaskB and linking itself to section TS_A_2. Section TS_A_2 ends with an action

TerminateTask. Each section TS_A_1 and TS_A_2 needs computation time to be estimated by WCET values. However, there is no deadline set for both of these sections.

TaskB consists of one section TS_B, which ends with an action *TerminateTask* after the task generates response. Task section TS_B has the deadline relative to the start of subtransaction (to the start of section TS_A_1 indeed).

Execution scenario of the subtransaction depends on the priorities of tasks. If TaskB has higher priority than TaskA, then the execution scenario is the sequence of sections TS_A_1, TS_B, TS_A_1. In this case deadline is met (**Figure 6.4**). However, if TaskB has lower priority than TaskA, then the execution scenario is the sequence of sections TS_A_1, TS_A_2, TS_B, and the deadline does not met (**Figure 6.5**).

Therefore, execution scenario depends on rescheduling of tasks. DS-Design Tool takes into consideration the rescheduling points to provide evaluation of response times.

From rescheduling points DS-Design Tool also learns how the tasks sections compete with other tasks on processor. Task sections are dependent from each other within subtransactions, but they are executed independently from the tasks, belonging to other transactions and independent subtransactions. Therefore, each time the execution scenario experiences rescheduling point, DS-Design Tool takes into consideration changes in the scheduler lists, and, hence, apply analysis to the changed lists.

NOTE DS-Design Tool extracts the information about scheduling from ACTION parameter in the description of the task section and ISR section. Each occurrence of the system service, which impacts scheduling should be framed by the task or the ISR section.

Sections and Processing Levels (priorities)

The priority of the tasks and interrupt service routines definitely affect the allocation of processor. Low priority tasks are preempted from processor by tasks of high priorities, thus delaying responses generated by the tasks of lower priorities. Therefore, DS-Design Tool takes into consideration the priorities of the tasks and interrupt services routines.

In spite that OSEKturbo employs fixed-priority scheduling, the priority is changed during the execution of the tasks and ISRs. That happens when the task or the ISR accesses the *critical sections* by means of using the OSEKturbo *resources*, or by means of disabling and suspending interrupts. Therefore, the information about boosting priorities in an application should be provided for DS-Design Tool. Task section and ISR section are used to provide this information.

NOTE DS-Design Tool extracts the information about effective priority of the task and the ISR from ACTION parameter in the description of the task section and the ISR section. Each occurrence of the system service that impacts the effective priority should be framed by the task or the ISR section.

NOTE To analyze scheduling of tasks that use OSEKturbo internal resources, the DS-Design Tool uses standard parameters of OIL. There is no need to describe task sections for internal resources.

Sections and Conditional Operators

Task sections and ISR sections may be used to describe the control flow within the task and the ISR. For example, in **Figure 6.6** three tasks are used to generate two different responses to one event depending on dynamically changed condition. TaskA is activated when event arrives (e.g. when alarm is expired). TaskA calculates the value of variable *x*, and activates TaskB if the value is positive. Otherwise, TaskC is activated. TaskB responds to the event with the response1, while TaskC responds with the response2. Both responses may have different deadlines - D1 and D2 correspondingly. TaskB and TaskC are of higher priority than TaskA.

There are two versions of execution scenario for this example. Execution scenario for positive value of *x* is presented on **Figure 6.7**. Execution scenario for non-positive value of *x* is presented on **Figure 6.8**. Surely, for each release of an event only one version of the execution scenario is implemented, so these scenarios are mutually exclusive. DS-Design Tool takes into consideration this fact when it calculates response times R1 for response1 and R2 for response2.

Figure 6.6 Handling if-then-else operators (scheme)

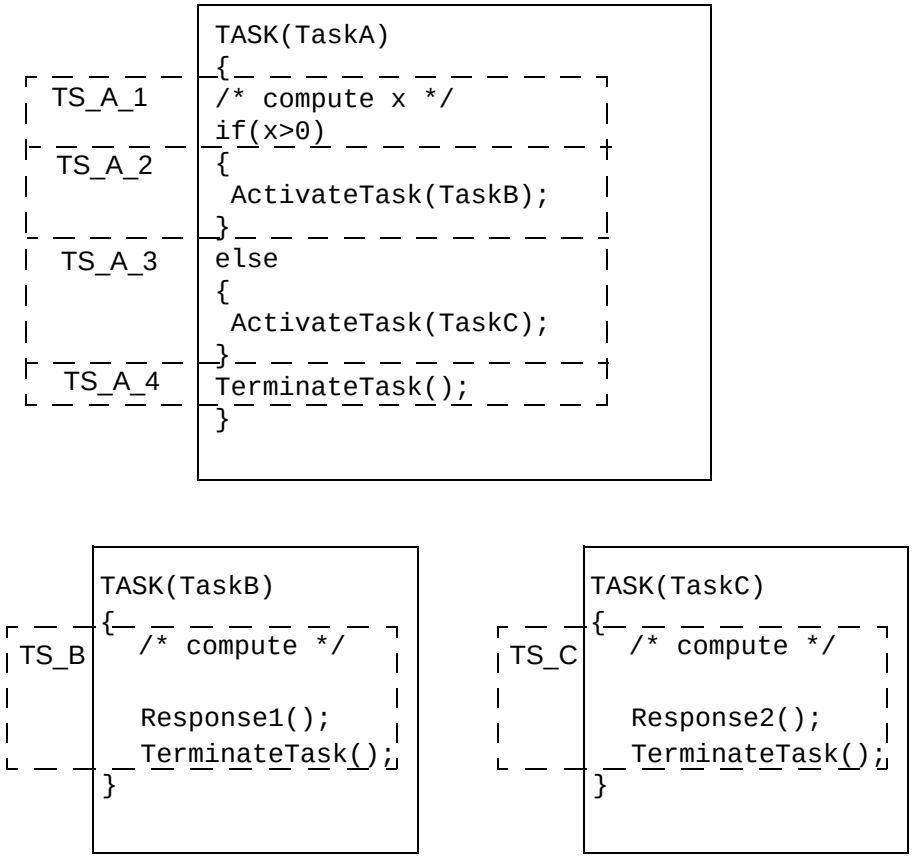


Figure 6.7 Handling if-then-else operators (response1)

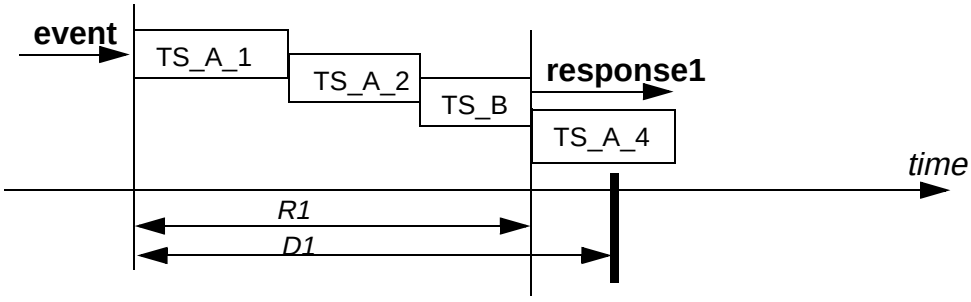
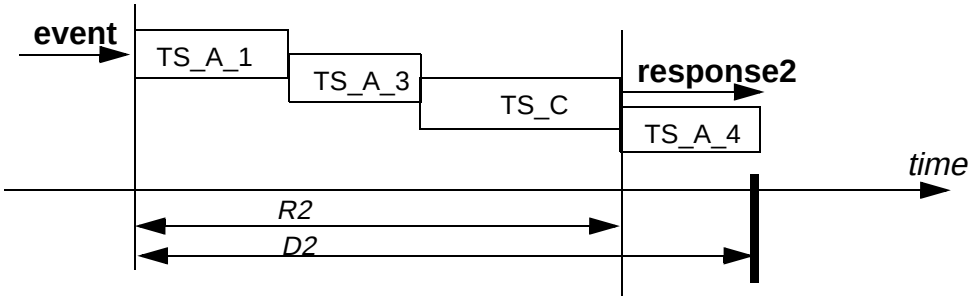


Figure 6.8 Handling if-then-else operators (response2)



An action called *GoTo* action is used in section TS_A_1 to describe the jump to section TS_A_2 and section TS_A_3. For each jump separate *GoTo* action is used.

NOTE GoTo actions may be used for conditional operators such as if-then-else and switch.

Actions Summary

Table 6.1 Task and ISR sections actions contains the full list of actions applicable for the task and the ISR sections.

Table 6.1 Task and ISR sections actions

Action	Description	TASK Section	ISR section
ActivateTask	Section calls OS service *ActivateTask*. Reference to the task that is being activated should be provided	+[a]	+
TerminateTask	Task section calls OS service *TerminateTask*. This is default action for task section. There is no next task section, because task goes into suspended state	+	-

Table 6.1 Task and ISR sections actions

Action	Description	TASK Section	ISR section
ChainTask	Task section calls OS service *ChainTask*. Reference to the task that is being activated should be provided. There is no next task section, because task goes into suspended state	+	-
Schedule	Task section calls OS service *Schedule*	+	-
DisableAllInterrupts	Section calls OS service DisableAllInterrupts	+	+
EnableAllInterrupts	Section calls OS service EnableAllInterrupts	+	+
SuspendAllInterrupts	Section calls OS service SuspendAllInterrupts	+	+
ResumeAllInterrupts	Section calls OS service ResumeAllInterrupts	+	+
SuspendOSInterrupts	Section calls OS service SuspendOSInterrupts	+	+
ResumeOSInterrupts	Section calls OS service ResumeOSInterrupts	+	+
GetResource	Section calls OS service *GetResource*. Reference to the resource that is being acquired should be provided	+	+[b]
ReleaseResource	Section calls OS service *ReleaseResource*. Reference to the resource that is being released should be provided	+	+[c]
SetEvent[d]	Section calls OS service *SetEvent.* Reference to the events that are being set should be provided	+	+
WaitEvent[e]	Task section calls OS service *WaitEvent*. Reference to the events that the task waits should be provided	+	-

Table 6.1 Task and ISR sections actions

Action	Description	TASK Section	ISR section
CheckEvent[f]	Task section checks the state of an event and jumps to the next task section depending on the state of the event (if it is set or cleared)	+	-
SendMessage	Section calls OS service *SendMessage*. Reference to the message that is being sent should be provided.	+	+
GoTo	Section jumps to the next task section	+	+
LeaveISR	ISR section calls OS service *LeaveISR*. This is default action for ISR object. There is no next ISR section, because ISR terminates	-	+

a. "+" means the action is valid, "-" means the action is not applicable

b. Applicable if OSEKturbo configured to support extension of resources for interrupt levels

c. Applicable if OSEKturbo configured to support extension of resources for interrupt levels

d. See "Analysis of ECC Applications" for explanation of this action

e. See "Analysis of ECC Applications" for explanation of this action

f. See "Analysis of ECC Applications" for explanation of this action

Timepoints and Checkpoints

Timepoints and Checkpoints Overview

Each termination of the task section or ISR section is called *`timepoint`*. Therefore, each task section and ISR section has a timepoint. At the timepoint the section **may** generate a response.

Not all timepoints are interesting for the analysis. As responses in real-time applications should meet deadlines, the most important timepoints are those where the responses should be completed before deadline. These timepoints are called checkpoints. *`Checkpoint`* is the timepoint for which deadline is set.

NOTE DS-Design Tool computes response times **only** for checkpoints. This is done in order to decrease the amount of calculations performed during analysis of an application.

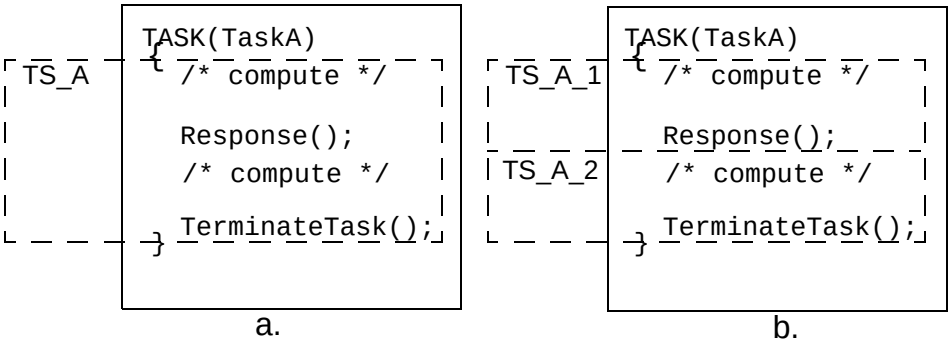
It is easy to convert the timepoint into the checkpoint and vise versa by means of setting and removing deadline parameter in the task or the ISR section.

Checking Deadlines in the Middle of Tasks and ISRs

In many cases in BCC applications the task section that terminates the task is checkpoint, because the task generates the response before termination.

However, when response is generated somewhere in the middle of the task, it is not obvious how to check the response time vs. deadline. In **Figure 6.9** on picture a. such task is shown. The problem may be solved by means of splitting the task section TS_A into two sections - TS_A_1 and TS_A_2 (picture b.). Section TS_A_1 ends with the generation of response, and uses action *GoTo* to tell the DS-Design Tool that task execution is continued in section TS_A_2. Checkpoint is set in section TS_A_1, and DS-Design Tool computes actual response time for this section, thus checking deadline.

Figure 6.9 Splitting the task section



NOTE GoTo action is a handy split tool for the task and ISR sections.

Execution Paths

Using Paths for Dependent Conditional Operators

Handling conditional operators by means of using task sections or ISR sections with *GoTo* actions as described in **Sections and Conditional Operators** is useful, but limited.

In many cases the branches of conditional operators are dependent of each other. The example of such algorithm is shown on **Figure 6.10**.

Figure 6.10 Dependent conditional operators (scheme)

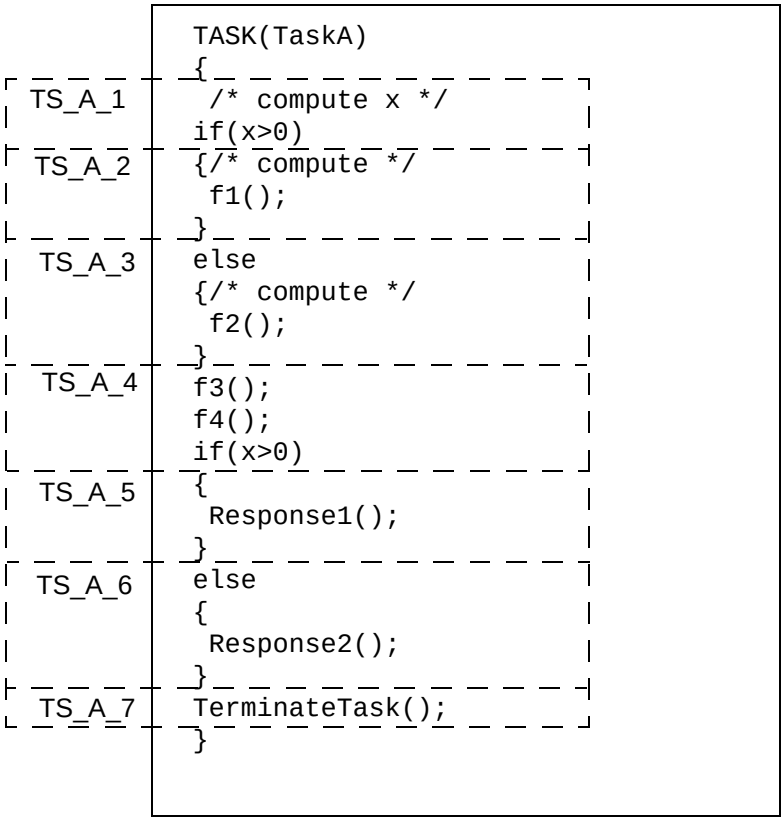


Figure 6.11 Dependent conditional operators (Path1)

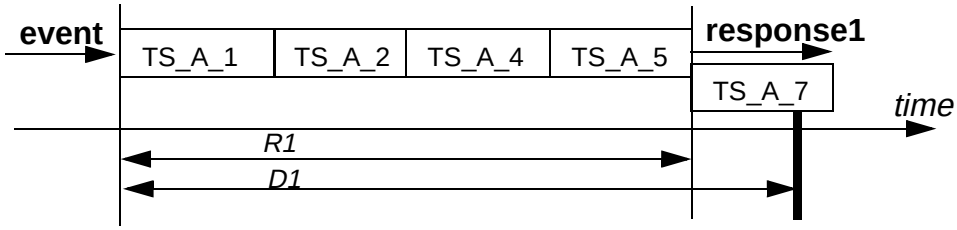
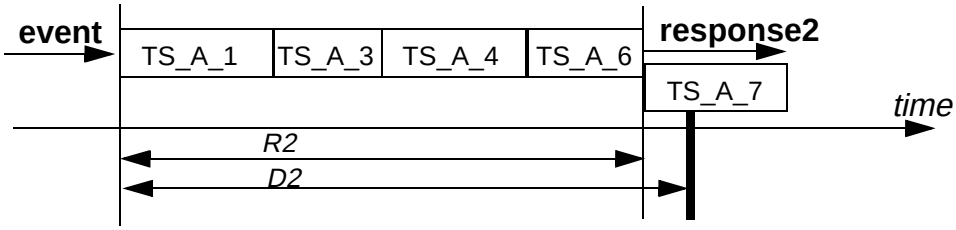


Figure 6.12 Dependent conditional operators (Path2)



TaskA is activated when an event arrives, and it generates two responses depending of the value of variable *x*. If value of *x* is positive, then response1 is generated which has deadline D1. If value of *x* is non-positive, then task generates response2 which has deadline D2 (**Figure 6.10**).

Tasks uses two conditional operators if-then-else to compute responses. Note that there is common computational block in between these conditional operators - the task section TS_A_4. The dependency between conditional operators is as following:

- if in first operator TS_A_2 is executed, then in second operator TS_A_5 is executed
- if in first operator TS_A_3 is executed, then in second operator TS_A_6 is executed

Therefore, for positive value of *x* the sequence of sections is as shown on **Figure 6.11**, while non-positive value of *x* leads to the sequence shown on **Figure 6.12**. Note that the middle common section TS_A_4 is executed in both routes.

These two routes, or paths of execution graph represent two execution scenarios. Task sections TS_A_2 and TS_A_5 belong to one path (Path1),

task sections TS_A_3 and TS_A_6 belong to another path (Path2), while sections TS_A_1, TS_A_4, and TS_A_7 belong to both paths.

NOTE Execution paths represent the algorithms of the application code. Execution paths allow more accurate analysis and provide less pessimistic results.
Path name is a unique symbolic name within the scope of the subtransaction.

Paths mechanism effectively remove unrealistic execution scenarios, thus decreasing pessimism of analysis. Indeed, if no paths are specified in this example, DS-Design Tool creates four execution scenarios:

```
1. TS_A_1, TS_A_2, TS_A_4, TS_A_5, TS_A_7 (realistic)
2. TS_A_1, TS_A_2, TS_A_4, TS_A_6, TS_A_7 (non realistic)
3. TS_A_1, TS_A_3, TS_A_4, TS_A_6, TS_A_7 (realistic)
4. TS_A_1, TS_A_3, TS_A_4, TS_A_5, TS_A_7 (non-realistic)
```

Scenarios 2 and 3 never happen as they go mutually exclusive branches of the first and the second if-then-else operators. The Paths mechanism allows exclude non-realistic scenarios from analysis.

Each action in the task section or in the ISR section may specify the path to which the action applies. If the task section belongs to more than one path, it defines action for each path. In the example on **Figure 6.10** each of task sections TS_A_1 and TS_A_4 has two *GoTo* actions - the first for Path1 and the second for Path2.

NOTE DS-Design Tool uses parameters PATH in the description of the task sections and the ISR sections to learn to which path(s) the action belongs to.

Using Paths Across Tasks and ISRs

The the other limitation of usage of *GoTo* actions as it is described in **Sections and Conditional Operators** there is the boundary of execution scenario, which is only within single task or single ISR.

In some cases conditions are spread over the tasks or the ISRs within subtransactions. DS-Design Tool applies execution paths approach to describe such conditions.

On **Figure 6.13** the event is processed by the chain of two tasks. TaskA is activated when the event arrives and generates response. The amount of the computations needed to generate the response, and the deadline of the response depends on the value of variable *x*. If value of *x* is positive, than TaskA makes short computation in section TS_A_2, and chains TaskB that completes the computation and generate response that should meet deadline D1. If value of *x* is non-positive, than TaskA makes longer computation in section TS_A_3, and chains TaskB that completes the computation and generate response that should meet longer deadline D2.

The scheme of tasks sections is shown on **Figure 6.13**, and execution scenarios for responses are presented on **Figure 6.14** and **Figure 6.15**.

Note that deadline defined in TaskB in section TS_B depends on conditions calculated in TaskA. To provide information about this condition, two paths are defined in an application - Path1 for positive value of *x* and for shorter deadline D1, and Path2 for non-positive value of *x*, and for longer deadline D2.

The information about paths is defined in action parameters of task sections of TaskA, and in the description of deadline in task section of TaskC thus allowing tracking of path to which deadline belongs.

NOTE DS-Design Tool uses parameters PATH in the description of the deadline of the task sections and the ISR sections to learn which path(s) the deadline belongs to.

Figure 6.13 Conditions dependent deadline (scheme)

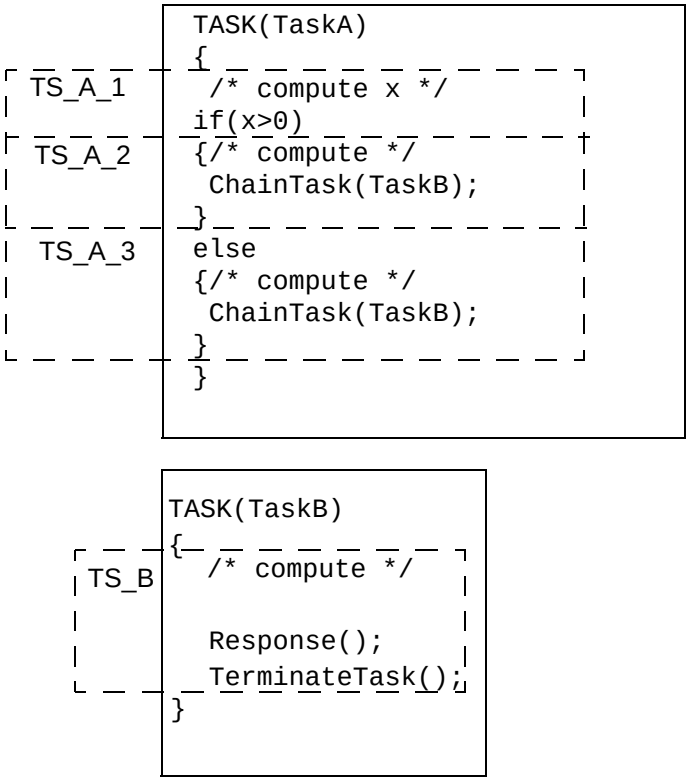


Figure 6.14 Conditions dependent deadline (Path1)

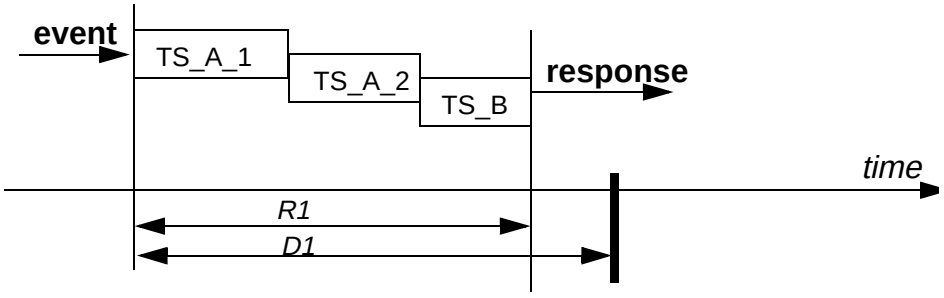
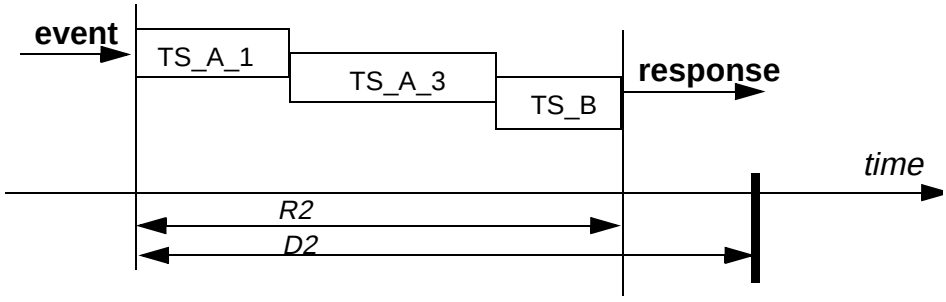


Figure 6.15 Conditions dependent deadline (Path2)



NOTE Clear design of an application helps to avoid excessive use of path features.

As paths links the tasks sections and the ISR sections of different tasks and ISRs within subtransaction, it may be used to follow the same conditional computations across tasks and ISRs.

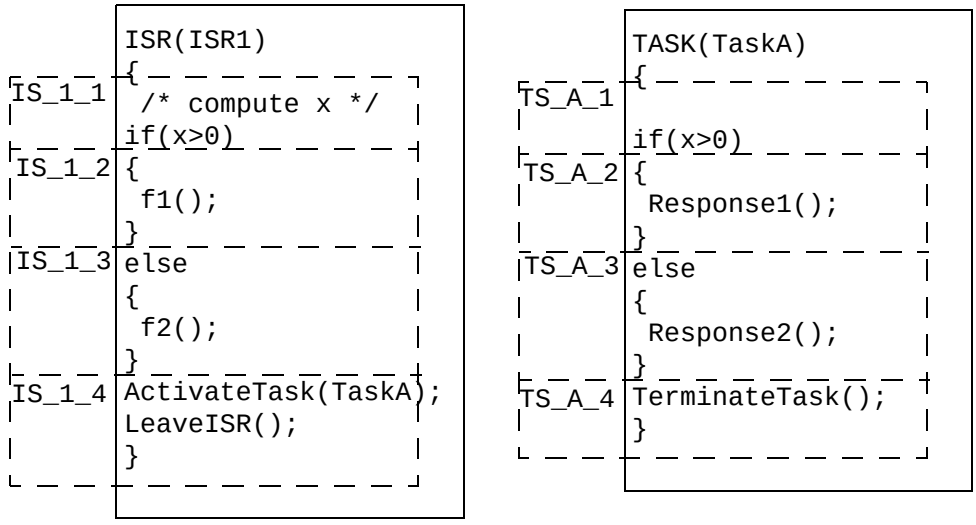
Figure 6.16 shows the subtransaction that uses same conditional variable in ISR and task. ISR1 is released when an event arrives. ISR1 computes the value of **global** variable *x*, and depending on the value of *x* perform different computations. Then ISR1 activates task TaskA to complete processing of an event and to generate two different responses depending on value of variable *x*. In both ISR1 and TaskA same condition is checked - the value of *x*.

In order to follow the condition from ISR1 in TaskA, two paths are created:

- Path1 for positive value of *x* and response1, comprises ISR sections IS_1_1, IS_1_2, IS_1_4, and task sections TS_A_1, TS_A_2, TS_A_4
- Path2 for non-positive value of *x* and response2, comprises ISR sections IS_1_1, IS_1_3, IS_1_4, and task sections TS_A_1, TS_A_3, TS_A_4

Hence, two execution scenario are analyzed by DS-Design Tool.

Figure 6.16 Same conditional computations across in ISR and task



Applying Computational Model

Building an Application Model

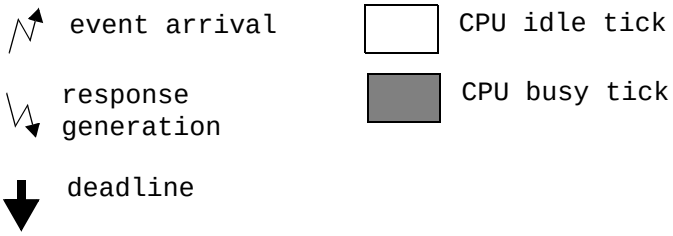
The following steps should be done by an application designer to apply schedulability analysis to the application:

1. Describe timing requirements to the application.
2. Identify transactions in an application.
3. Identify subtransactions in an application.
4. Sketch execution graph of each subtransaction.
5. Build execution scenarios of each subtransaction.
6. Identify the tasks and the ISRs sections that each execution snaring comprises.
7. Identify timepoints and checkpoints.
8. Describe the tasks and ISRs sections in an OIL file (by means of using OSEK Builder or text editor).

9. Run DS-Design Tool and perform the schedulability analysis.
10. Explore results of analysis.

Subsections below demonstrate the use of these steps for sample applications.

The following legend is used for timing diagrams:

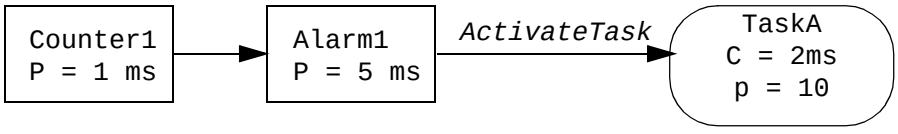


WARNING! For simplification reasons in the examples below the operating system overhead is not considered.
The examples are intentionally simple for the sake of making explanations clear. Real applications have much more entities, and usually computational times are much less than deadlines.

The graphical images in this document may slightly differ from DS-Design Tool interface. The difference does not impact functionality of DS-Design Tool.

Single Periodic Task

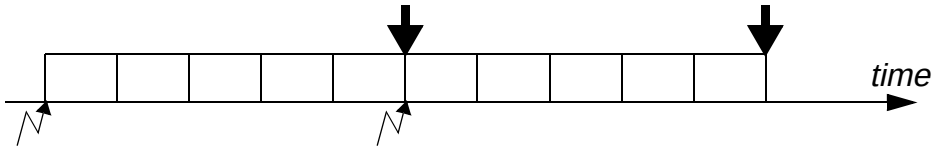
The application consists of the single task *TaskA* of priority 10 that is activated periodically by means of using an alarm *Alarm1* connected to the timer *Counter1* as presented on the picture below[1]:



[1] Here and in examples below "P" stands for *period*, "C" stands for *computation time*, and "p" stands for *priority*.

The counter period is 1 milliseconds (ms), the alarm is cyclically activated every 5 ms, i.e. every 5th tick of the counter. Task should generate the response within 5 ms, using 2 ms of the processor time. Task generates response immediately before termination.

Step 1. Timing requirements of applications are presented on the picture:



Event arrives every 5 ms, and should be processed before deadline that is 5 ms from the arrival of the event.

Step2. Application has single transaction *Counter1* that has period 1 ms.

Step3. Transaction *Counter1* has single subtransaction *Alarm1* (full name *Counter1/Alarm1*).

Step 4. The execution graph of subtransaction is just *TaskA*, or, more precisely, single task section of *TaskA*, identified as *TS_A*.

Step 5. There is only one execution scenario of the subtransaction: task section *TS_A* of *TaskA*.

Step 6. Task section *TS_A* ends with an action *TerminateTask*. Worst-case execution time of the section is 2 ms.

Step 7. Single checkpoint is at the end of *TaskA*, when the task generates response and terminates. Full name of checkpoint is *Counter1/Alarm1/ TaskA/TS_A*. The deadline is 5 ms.

Step 8.OIL file for the example is presented below. The DS-Design Tool specific parameters are in **bold**.

```
// SinglePeriodicTask.oil

OIL_VERSION = "2.3";
#include "ost22mpc.oil"

CPU cpu1
{
    APPMODE app1;
```

```
    OS os1
    {
      TargetMCU = MPC555;
      STATUS = STANDARD;
      ResourceScheduler = FALSE;
      STARTUPHOOK = FALSE;SHUTDOWNHOOK = FALSE;
      PRETASKHOOK = FALSE;POSTTASKHOOK = FALSE;ERRORHOOK = FALSE;
      USEGETSERVICEID = FALSE; USEPARAMETERACCESS = FALSE;

      SATimeUnit = ms;
      saMeasureClockFrequency = 4000;
    };

    TASK TaskA
    {
      PRIORITY = 10;
      SCHEDULE = FULL;
      AUTOSTART = FALSE;
      ACTIVATION = 1;

      saTaskSection = SET { ID = "TS_A";
          WCET = 2;
          DEADLINE = SET { VALUE = 5; };
          ACTION = TerminateTask;
      };
    };

    COUNTER Counter1
    {
      MAXALLOWEDVALUE = 0xffffffff; MINCYCLE = 0; TICKSPERBASE = 1;
      saPeriod = 1;
    };

    ALARM Alarm1
    {
      COUNTER = Counter1;
      ACTION = ACTIVATETASK { TASK = TaskA; };
      AUTOSTART = FALSE;
      saCyclic = TRUE { ALARMTIME = 0; CYCLETIME = 5; };
    };
};
```

In this example the alarm is not autostarted. That means it should be started in the application code by means of calling *SetRelAlarm* system service.

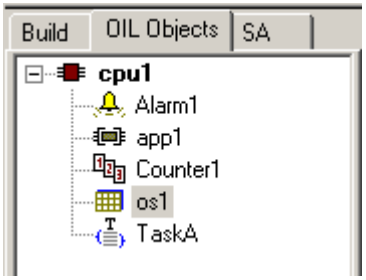
In order to avoid explicit start of alarm in the code, the alarm may be converted into autostart alarm. For the example in the listing above this can be done by means of changing two lines in the description

of *Alarm1*:

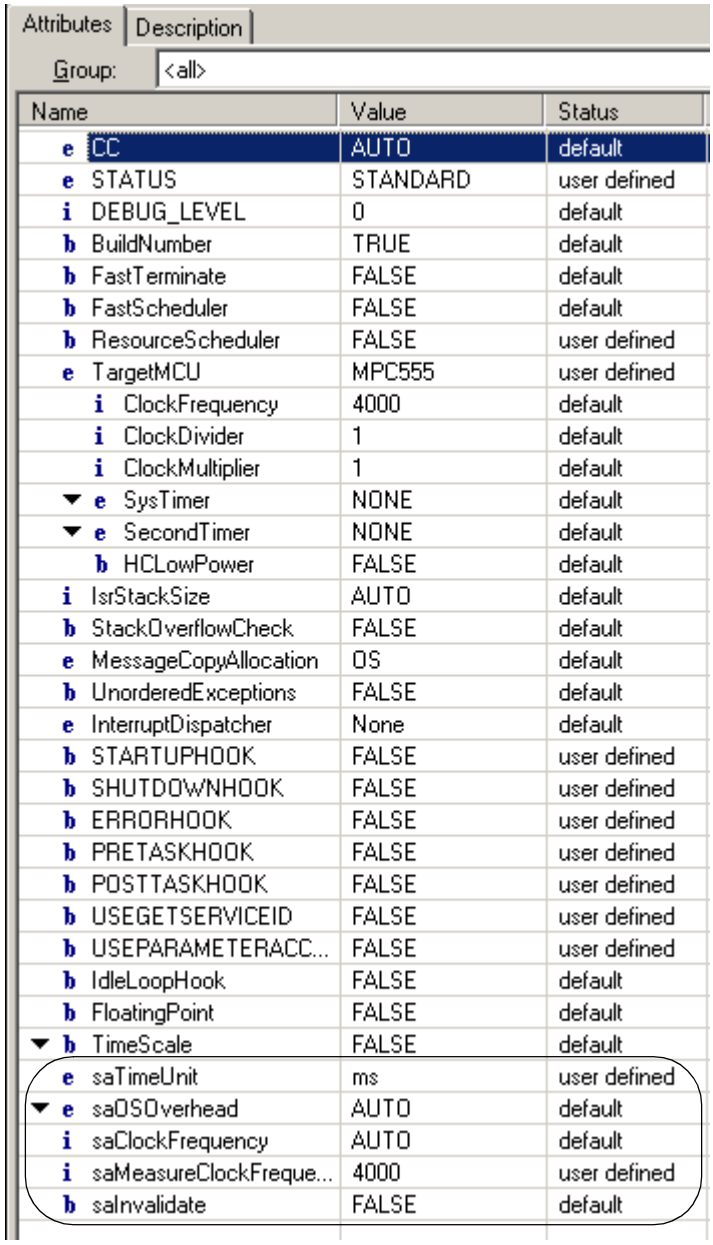
```
ALARM Alarm1
{
  COUNTER = Counter1;
  ACTION = ACTIVATETASK { TASK = TaskA; };
  AUTOSTART = TRUE { ALARMTIME = 0; CYCLETIME = 5; };
};
```

Note that *AUTOSTART* parameter in *ALARM* is not DS-Design Tool specific, but it is used by DS-Design Tool to learn the alarm behavior.
This method is applicable for next examples as well.

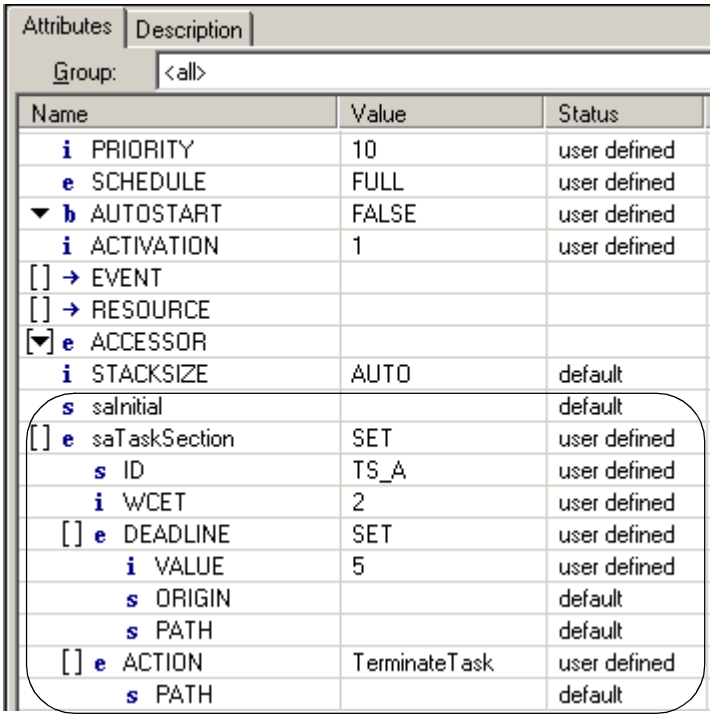
In OSEK Builder the pane of OIL objects should include the following ones:



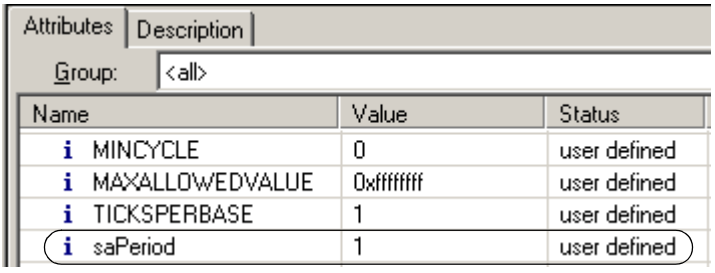
Parameters of *os1* are presented below (DS-Design Tool specific parameters are framed):



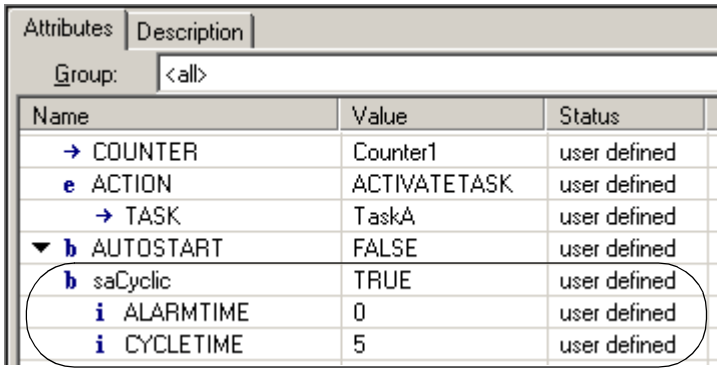
Parameters of *TaskA* are presented below (DS-Design Tool specific parameters are framed):



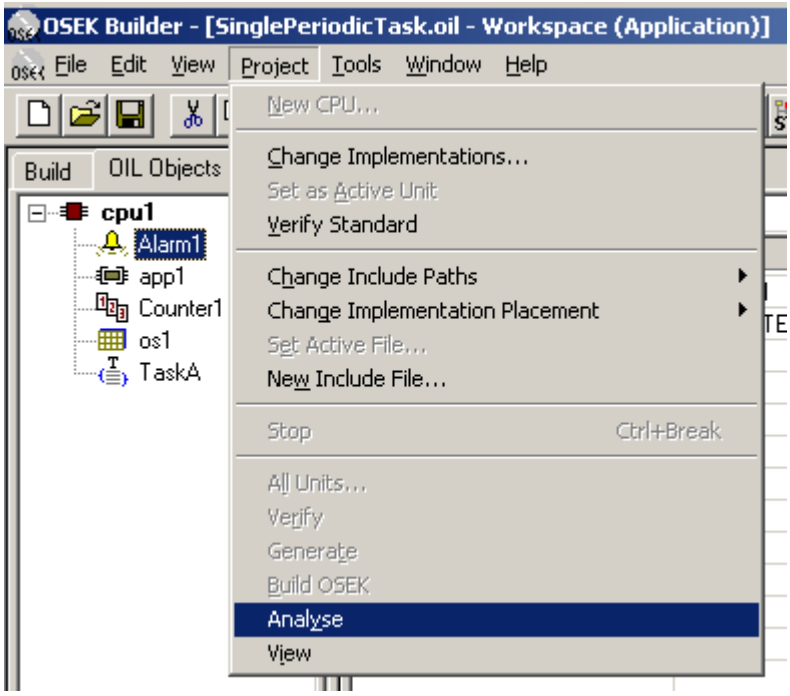
Parameters of *Counter1* are presented below (DS-Design Tool specific parameters are framed):



Parameters of *Alarm1* are presented below (DS-Design Tool specific parameters are framed):



Step 9. DS-Design Tool should be launched from OSEK Builder using the main menu:



1. Analysis is started by means of clicking menu item Project/ Analyze.
2. Graphical views tool is started by means of clicking menu item Project/View.

Alternatively, OSEK Builder toolbar buttons may be used:

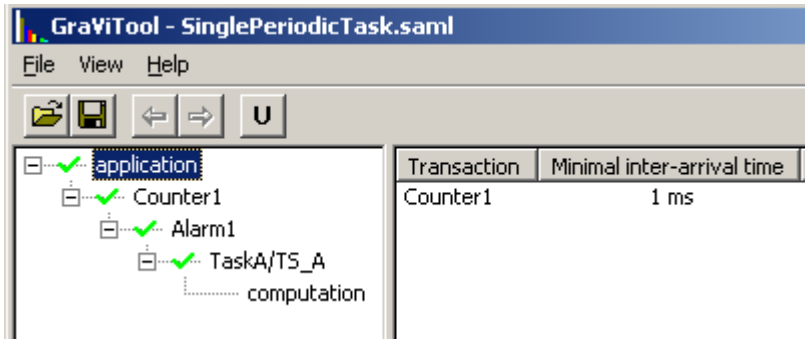
1. Analysis is started by means of clicking button 1.
2. Graphical views tool is started by means of clicking button 2.

During the analysis phase, the output window of OSEK Builder displays error messages.

Step 10. Results of analysis are displayed in GraViTool panes.

The left pane displays transactions, subtransactions, and checkpoints as a tree. The right pane contains details of the object highlighted on the left pane on the tree.

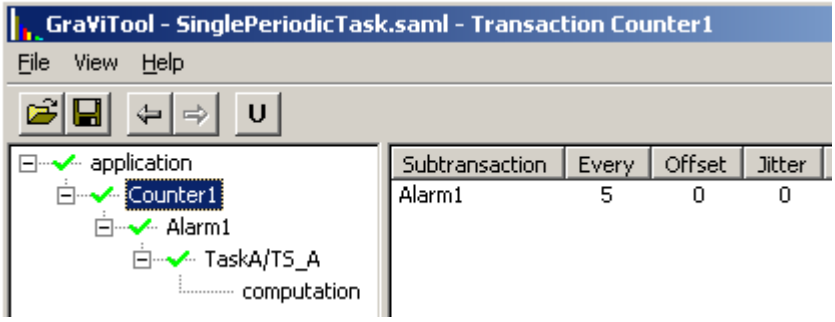
If *application* is selected, then the right pane displays the list of transactions. In the example it is *Counter1* with the period 1 ms. Note that transaction is created automatically - it is not described in OIL file.



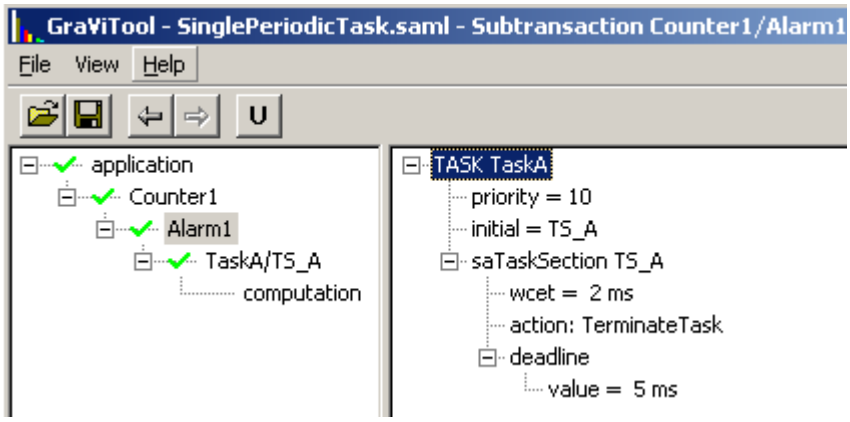
NOTE Green vote mark on the tree branch means that corresponding checkpoint meets deadline.

Red scratch mark on the tree branch means than corresponding checkpoint does not meet deadline.

If transaction is selected, then the right pane contains description of the selected subtransactions. In the example there is only one periodic subtransaction *Counter1/Alarm1*, that has the period 5 counter ticks. Note, that subtransaction is created automatically - it is not described in OIL file:

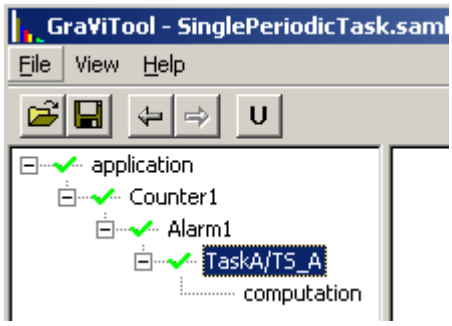


When subtransaction *Counter1/Alarm1* is selected, the right pane displays the sections of the subtransaction. All details of sections are displayed in a tree-view:

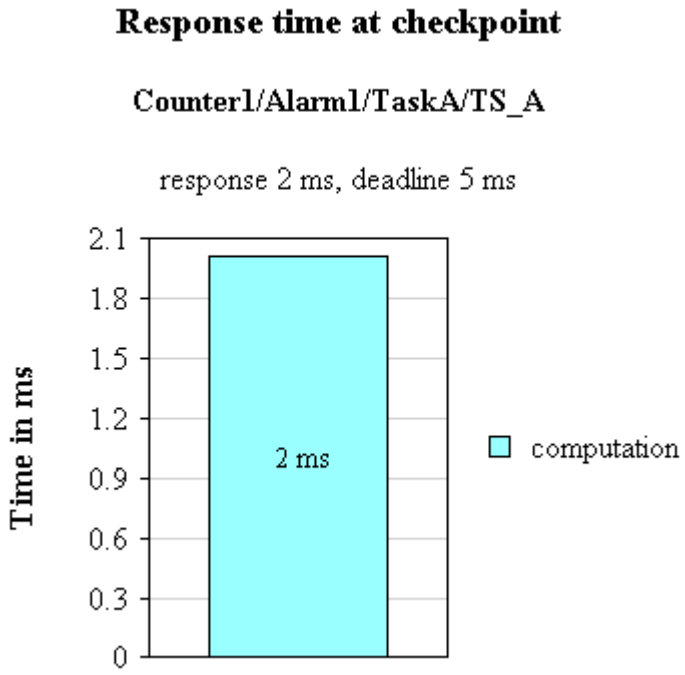


The sections' tree allows to check if OIL description is consistent and really reflects an application design.

For the checkpoint selected as it is shown below

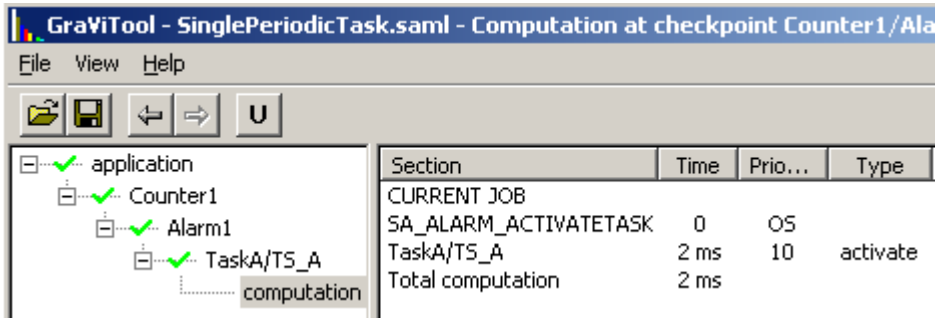


the right pane displays the response time:



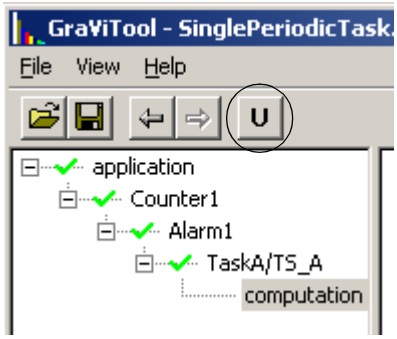
The pane shows that checkpoint meets deadline as response time is 2 ms while deadline is 5 ms. This is the trivial result as only one task occupies the processor.

Click on the computation bar or on the computation branch in the tree brings the table that reflects the breakdown of the response time:

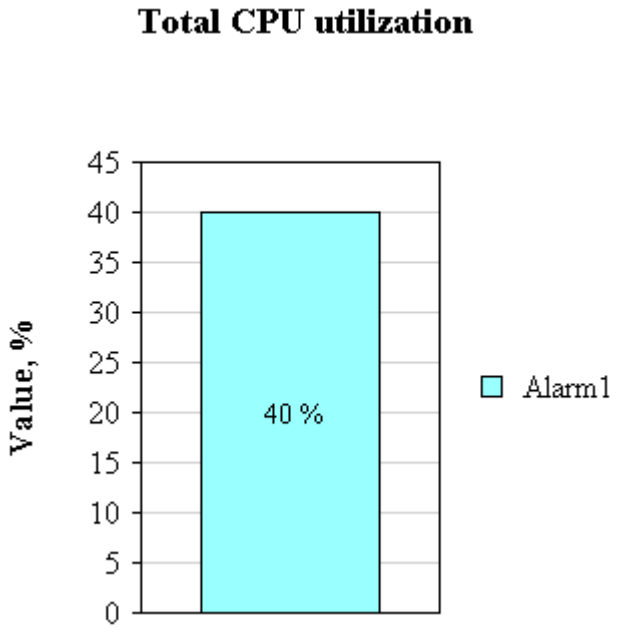


Again, in this example the only contribution into response time does task section *TS_A* of *TaskA*, which is executed with priority value 10.

The tool also shows the utilization of processor by the application. Click on toolbar button *U*:



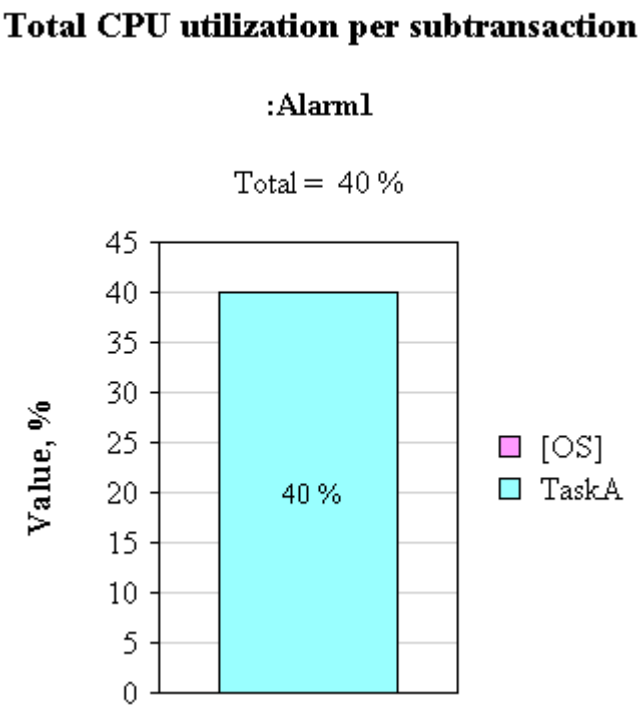
brings the utilization chart in right pane:



In the example the total utilization of CPU is 40% (actually it is easy to calculate by hands as it is 2 ms of execution time divided by 5 ms of event arrival).

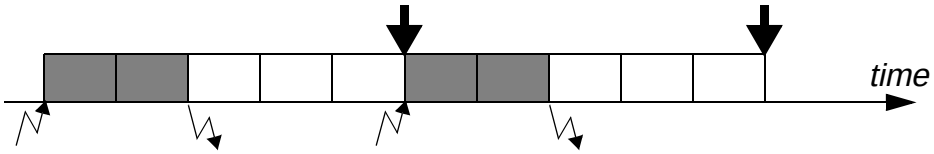
The total utilization chart shows contribution of each subtransaction into processor utilization. Click on the bar of the subtransaction utilization, bar

brings the chart, which shows contribution of each subtransaction scenario task or ISR into the utilization share of the subtransaction:



In the example the only contributor into execution scenario is *TaskA*.

The analysis results of this example may be explained by means of sketching a worst-case application scenario by hands:



Each 5 ms the event is processed by the CPU that spends 2 ms for response generation. As there is no other events in the application, the processing of an event starts immediately as the event arrives, and it is not preempted. Therefore, each response is generated in 2 ms well before deadline.

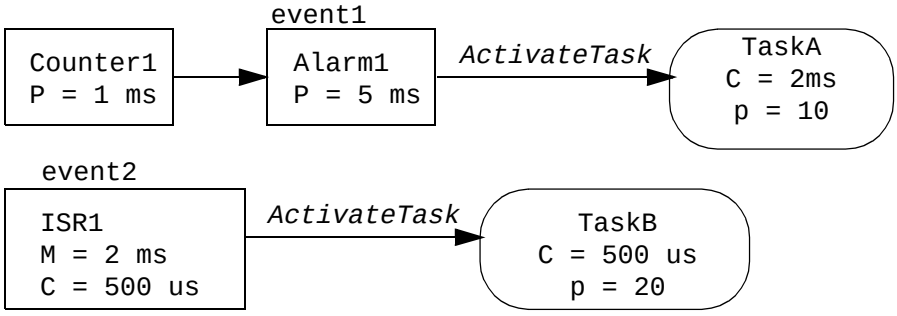
DS-Design Tool does not produce the worst-case execution scenario, because it is almost useless even for dozen tasks and ISRs in real applications.

In the next examples application configuration will be shown as OIL file, and graphical views will be shown briefly. For details of using OSEK Builder and graphical views please refer to the example in this subsection.

Periodic Task and an ISR

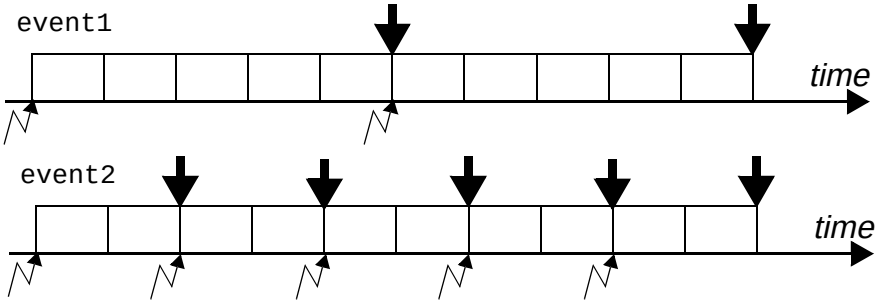
The example is modification of the example in subsection **Single Periodic Task**, therefore here only additional details are described.

In addition to periodic task an ISR is added. *ISR1* activates task *TaskB* which has priority 20:



The *ISR1* has minimal interarrival time 2 milliseconds (ms), and takes 500 microseconds (us) of computational time. Activated *TaskB* should generate the response within 2 ms, using 500 us of the processor time. *TaskB* generates response immediately before termination.

Step 1. Timing requirements of applications events are presented on the picture:



Event1 arrives every 5 ms, and should be processed before deadline that is 5 ms from the arrival of the event. *Event2* arrives every 2 ms, and should be processed before deadline that is 2 ms from the arrival of the event.

Step2. Application has two transactions:

- *Counter1* that has period 1 ms
- *ISR1* that has minimal interarrival time 2 ms

Step3. Application subtransactions:

- Transaction *Counter1* has single subtransaction *Alarm1* (full name *Counter1/Alarm1*)
- Transaction *ISR1* has single subtransaction *ISR1* (full name *ISR1/ ISR1*).

Step 4. The execution graphs:

- subtransaction *Counter1/Alarm1: TaskA* (more precisely single task section *TS_A*)
- subtransaction *ISR1/ISR1: ISR1* and *TaskB* (more precisely, two ISR sections of ISR1, identified as *IS_1_1* and *IS_1_2*, and single task section *TS_B*). ISR section *IS_1_1* is an initial section of *ISR1*

Step 5. There is only one execution scenario for each subtransactions, that is same as execution graph. Note, that both section of *ISR1* are executed before task section *TS_B*, as ISR1 has higher priority, than *TaskB*.

Step 6. Tasks and ISR sections are as following:

- *TaskA/TS_A*, that ends with an action *TerminateTask*. Worst-case execution time of the section is 2 ms
- *ISR1/IS_1_1*, that ends with an action *ActivateTask*, and is followed by the section *IS_1_2*. Worst-case execution time of the section is 300 us
- *ISR1/IS_1_2*, that ends with an action *LeaveISR*. Worst-case execution time of the section is 200 us
- *TaskB/TS_B*, that ends with an action *TerminateTask*. Worst-case execution time of the section is 500 us

Step 7. The checkpoints are as following:

- Checkpoint is at the end of *TaskA*, when the task generates response and terminates. Full name of checkpoint is *Counter1/Alarm1/TaskA/ TS_A*. The deadline is 5 ms

- Checkpoint is at the end of *TaskB*, when the task generates response and terminates. Full name of checkpoint is *ISR1/ISR1/TaskB/TS_B*. The deadline is 2 ms

Step 8.OIL file for the example is presented below. The DS-Design Tool specific parameters are in **bold**.

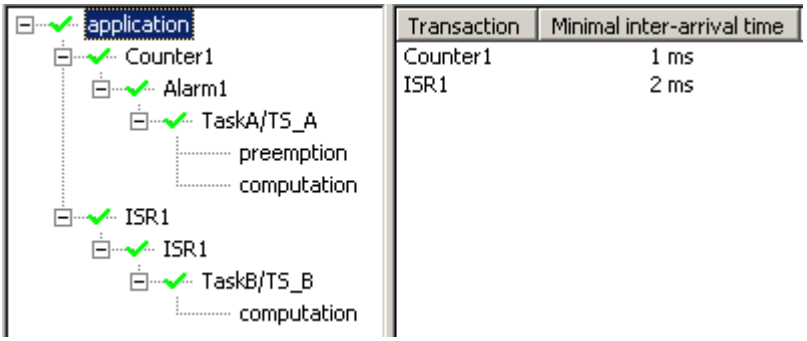
```
// PeriodicTaskAndISR.oil

OIL_VERSION = "2.3";
#include "C:\metrowerks\osek\ostmpc\bin\ost22mpc.oil"

CPU cpu1
{
    APPMODE app1;
    OS os1
    {
      TargetMCU = MPC555;
      STATUS = STANDARD;
      ResourceScheduler = FALSE;
      STARTUPHOOK = FALSE;SHUTDOWNHOOK = FALSE;
      PRETASKHOOK = FALSE;POSTTASKHOOK = FALSE;ERRORHOOK = FALSE;
      USEGETSERVICEID = FALSE; USEPARAMETERACCESS = FALSE;

      saTimeUnit = us;
      saMeasureClockFrequency = 4000;
    };

    TASK TaskA
    {
      PRIORITY = 10;
      SCHEDULE = FULL;
      AUTOSTART = FALSE;
      ACTIVATION = 1;

      saTaskSection = SET { ID = "TS_A";
          WCET = 2000;
          DEADLINE = SET { VALUE = 5000; };
          ACTION = TerminateTask;
      };
    };

    COUNTER Counter1
    {
      MAXALLOWEDVALUE = 0xffffffff; MINCYCLE = 0; TICKSPERBASE = 1;
      saPeriod = 1000;
    };

    ALARM Alarm1
    {
      COUNTER = Counter1;
```

```
    ACTION = ACTIVATETASK { TASK = TaskA; };
    AUTOSTART = FALSE;
    saCyclic = TRUE { ALARMTIME = 0; CYCLETIME = 5; };
  };

  TASK TaskB
  {
    PRIORITY = 20;
    SCHEDULE = FULL;
    AUTOSTART = FALSE;
    ACTIVATION = 1;

    saTaskSection = SET { ID = "TS_B";
        WCET = 500;
        DEADLINE = SET { VALUE = 2000; };
        ACTION = TerminateTask;
    };
  };

  ISR ISR1
  {
    CATEGORY = 2;
    PRIORITY = 10;

    saInitial="IS_1_1";
    saMinIATime = 2000;

    saISRSection = SET { ID = "IS_1_1";
        WCET = 300;
        ACTION = ActivateTask { NEXT = "IS_1_2"; TASK = TaskB; };
    };

    saISRSection = SET { ID = "IS_1_2";
        WCET = 200;
        ACTION = LeaveISR;
    };
  };
};
```

Step 9. Run DS-Design Tool.

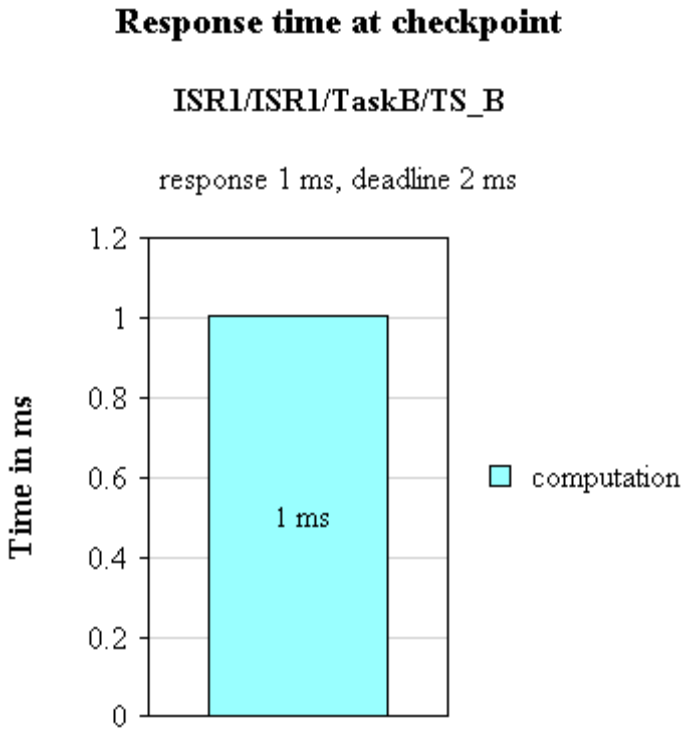
Step 10. The result of the analysis are explained below.

The transactions and checkpoints:



Quick look at green vote marks shows that both checkpoints are met.

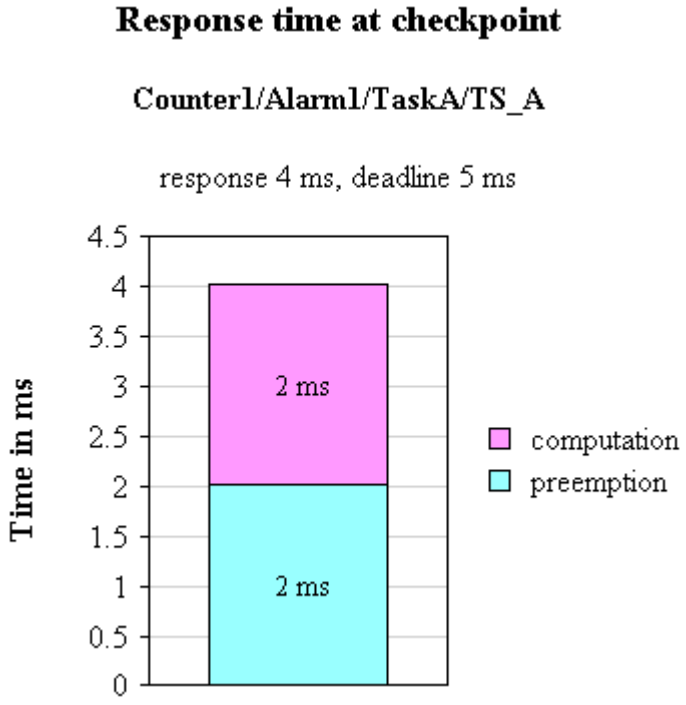
The details of checkpoint in *TaskB* is presented on picture:



The picture shows that response time is 1 ms, and click on the computation bar shows the distribution of computation time:

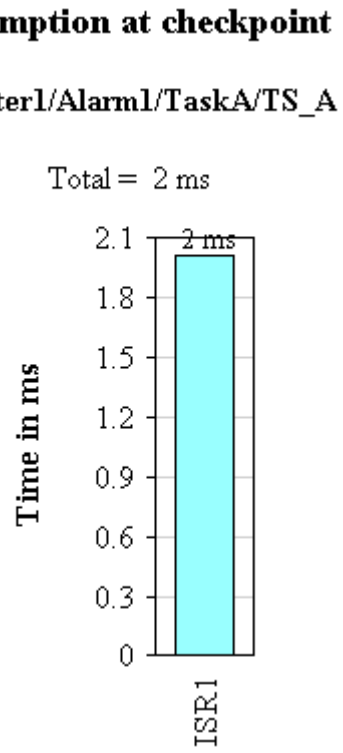
Section	Time	Priority	Type
CURRENT JOB			
SA_ENTER_ISR	0	OS	
ISR1/IS_1_1	300 µs	10	activate
SA_ISR_ACTIVATE_TASK	0	OS	
ISR1/IS_1_2	200 µs	10	return
SA_LEAVE_ISR	0	OS	
TaskB/TS_B	500 µs	20	activate
Total computation	1 ms		

Probably, more interesting is exploring the response at the end of *TaskA*:



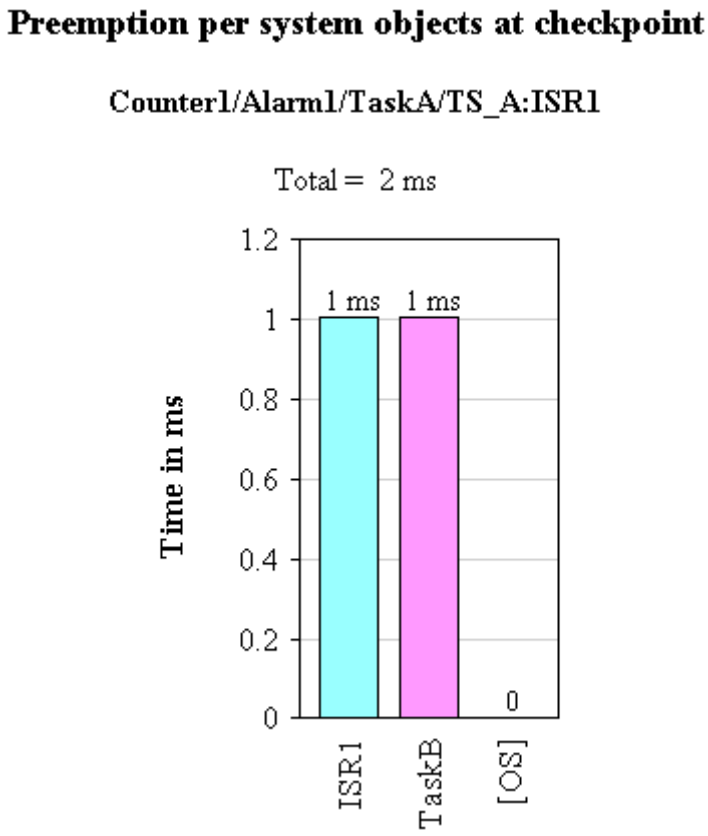
The response is generated in 4 ms - not 2 ms, as it is in previous example. Note, that the computation time is 2 ms, but the response is delayed by 2

ms due to preemption from entities of higher priority. Click on preemption bar brings the following chart:



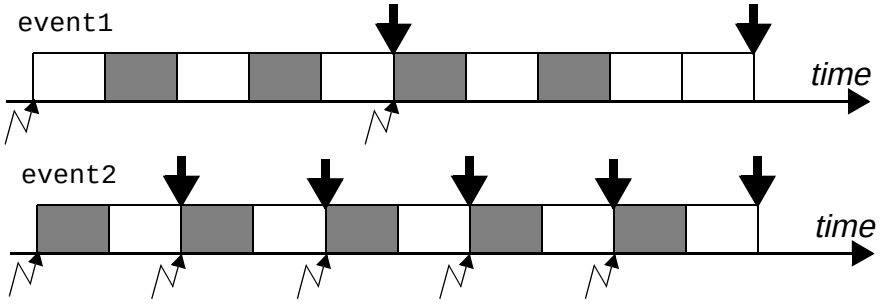
The chart shows that response is preempted by transaction *ISR1*. That is reasonable as ISR transaction consist of *ISR1* and *TaskB*, that both have

higher priority, than *TaskA*. Next click on *ISR1* bar brings the details of preemptions:



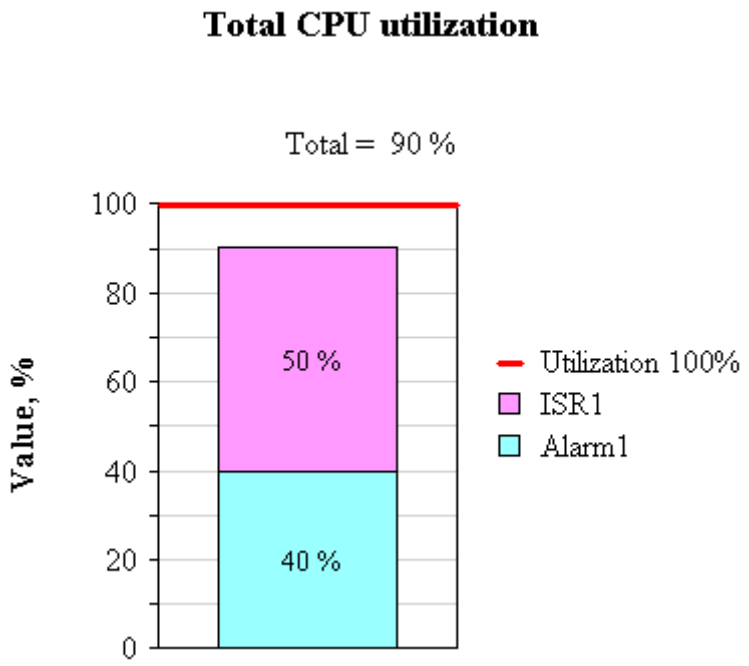
The chart shows, that while response to event2 is generated, *TaskA* experiences 1 ms preemption from interrupt service routine *ISR1*, and 1 ms preemption from task *TaskB*.

The preemption is explained by worst-case execution scenario, that may be sketched by hand:



Bottom diagram shows the execution of subtransaction *ISR1/ISR1*. Each filled box represent execution of *ISR1* and *TaskB* (500 us each). Top diagram shows the execution of *TaskA* of subtransaction *Counter1/Alarm1*. If both event1 and event2 occur simultaneously, then *TaskA* is preempted twice before it generates response.

Note, that processor is not fully utilized. Last box on diagram is idle. Therefore, CPU utilization is 90%, which is also shown in the utilization chart:



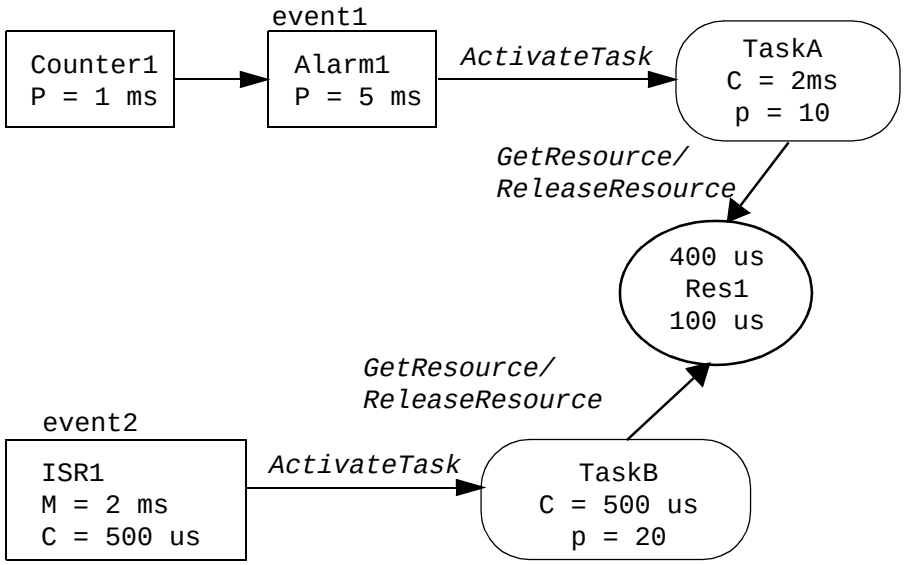
Half of CPU cycles are allocated to subtransaction *ISR1/ISR1*, while 40% are utilized by transaction *Counter1/Alarm1*.

NOTE The example demonstrates the utilization bound for independent periodic tasks explained in **Basic Methods of Schedulability Analysis**.

Periodic Task, an ISR and the Resource

The example is modification of the example in subsection **Periodic Task and an ISR**, therefore here only additional details are described.

Both *TaskA* and *TaskB* needs to access same hardware in order to generate response. As hardware module can not be accessed in a preemptive manner, it is considered as an OSEKturbo critical section, and is protected by *Res1*. *TaskA* needs 400 us of access to *Res1*, while *TaskB* needs 100 us. The scheme of the application is shown below:



Step 1, **Step 2**, and **Step3** are not changed.

Step 4. The execution graphs:

- subtransaction *Counter1/Alarm1: TaskA* (more precisely, there are task sections *TS_A_1*, *TS_A_2*, that delimit an access to resource *Res1*, and section *TS_A*). Task section *TS_A_1* is the initial section of *TaskA*
- subtransaction *ISR1/ISR1: ISR1* and *TaskB* (more precisely, two ISR sections of ISR1, identified as *IS_1_1* and *IS_1_2*, and task sections *TS_B_1*, *TS_B_2*, that delimit an access to resource *Res1*, and section *TS_B*). Task section *TS_B_1* is the initial section of *TaskB*

Step 5. There is only one execution scenario for each subtransactions, that is same as execution graph.

Step 6. Tasks and ISR sections are as following:

- *TaskA/TS_A_1*, that ends with an action *GetResource*. Worst-case execution time is 600 us
- *TaskA/TS_A_2*, that ends with an action *ReleaseResource*. Worst-case execution time is 400 us. This section is actually critical section

- *TaskA/TS_A*, that ends with an action *TerminateTask*. Worst-case execution time of the section is 1 ms
- *ISR1/IS_1_1*, that ends with an action *ActivateTask*, and is followed by the section *IS_1_2*. Worst-case execution time of the section is 300 us
- *ISR1/IS_1_2*, that ends with an action *LeaveISR*. Worst-case execution time of the section is 200 us
- *TaskB/TS_B_1*, that ends with an action *GetResource*. Worst-case execution time is 200 us
- *TaskB/TS_B_2*, that ends with an action *ReleaseResource*. Worst-case execution time is 100 us. This section is actually critical section
- *TaskB/TS_B*, that ends with an action *TerminateTask*. Worst-case execution time of the section is 200 us

Step 7. The checkpoints are not changed. Note, that **total** WCET of each subtransaction is not changed as well.

Step 8.OIL file for the example is presented below. The DS-Design Tool specific parameters are in **bold**.

```
// PeriodicTaskISRAndResource.oil

OIL_VERSION = "2.3";
#include "C:\metrowerks\osek\ostmpc\bin\ost22mpc.oil"

CPU cpu1
{
    APPMODE app1;
    OS os1
    {
      TargetMCU = MPC555;
      STATUS = STANDARD;
      ResourceScheduler = FALSE;
      STARTUPHOOK = FALSE;SHUTDOWNHOOK = FALSE;
      PRETASKHOOK = FALSE;POSTTASKHOOK = FALSE;ERRORHOOK = FALSE;
      USEGETSERVICEID = FALSE; USEPARAMETERACCESS = FALSE;

      saTimeUnit = us;
      saMeasureClockFrequency = 4000;
    };

    TASK TaskA
    {
      PRIORITY = 10;
      SCHEDULE = FULL;
      AUTOSTART = FALSE;
      ACTIVATION = 1;
```

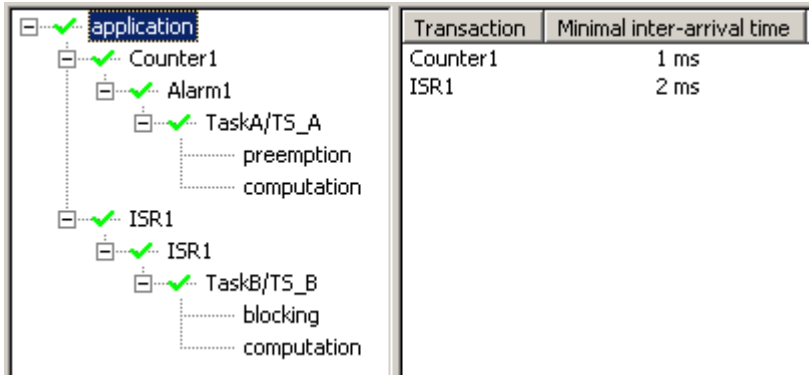
```
      RESOURCE = Res1;

      saInitial = "TS_A_1";
      saTaskSection = SET
      {
          ID = "TS_A_1";
          WCET = 600;
          ACTION = GetResource { NEXT = "TS_A_2"; RESOURCE = Res1;
};
      };
      saTaskSection = SET
      {
          ID = "TS_A_2";
          WCET = 400;
         ACTION = ReleaseResource { NEXT = "TS_A"; RESOURCE = Res1;
};
      };
      saTaskSection = SET { ID = "TS_A";
          WCET = 1000;
          DEADLINE = SET { VALUE = 5000; };
          ACTION = TerminateTask;
      };
    };

    COUNTER Counter1
    {
      MAXALLOWEDVALUE = 0xffffffff; MINCYCLE = 0; TICKSPERBASE = 1;
      saPeriod = 1000;
    };

    ALARM Alarm1
    {
      COUNTER = Counter1;
      ACTION = ACTIVATETASK { TASK = TaskA; };
      AUTOSTART = FALSE;
      saCyclic = TRUE { ALARMTIME = 0; CYCLETIME = 5; };
    };

    TASK TaskB
    {
      PRIORITY = 20;
      SCHEDULE = FULL;
      AUTOSTART = FALSE;
      ACTIVATION = 1;
      RESOURCE = Res1;

      saInitial = "TS_B_1";
      saTaskSection = SET
      {
          ID = "TS_B_1";
          WCET = 200;
          ACTION = GetResource { NEXT = "TS_B_2"; RESOURCE = Res1;
};
      };
```

```
      saTaskSection = SET
      {
          ID = "TS_B_2";
          WCET = 100;
         ACTION = ReleaseResource { NEXT = "TS_B"; RESOURCE = Res1;
};
      };
      saTaskSection = SET { ID = "TS_B";
          WCET = 200;
          DEADLINE = SET { VALUE = 2000; };
          ACTION = TerminateTask;
      };
    };

    ISR ISR1
    {
      CATEGORY = 2;
      PRIORITY = 10;

      saInitial="IS_1_1";
      saMinIATime = 2000;

      saISRSection = SET { ID = "IS_1_1";
          WCET = 300;
          ACTION = ActivateTask { NEXT = "IS_1_2"; TASK = TaskB; };
      };

      saISRSection = SET { ID = "IS_1_2";
          WCET = 200;
          ACTION = LeaveISR;
      };
    };

    RESOURCE Res1
    {
      RESOURCEPROPERTY = STANDARD;
    };
};
```

Step 9. Run DS-Design Tool.

Step 10. The result of the analysis is explained below.

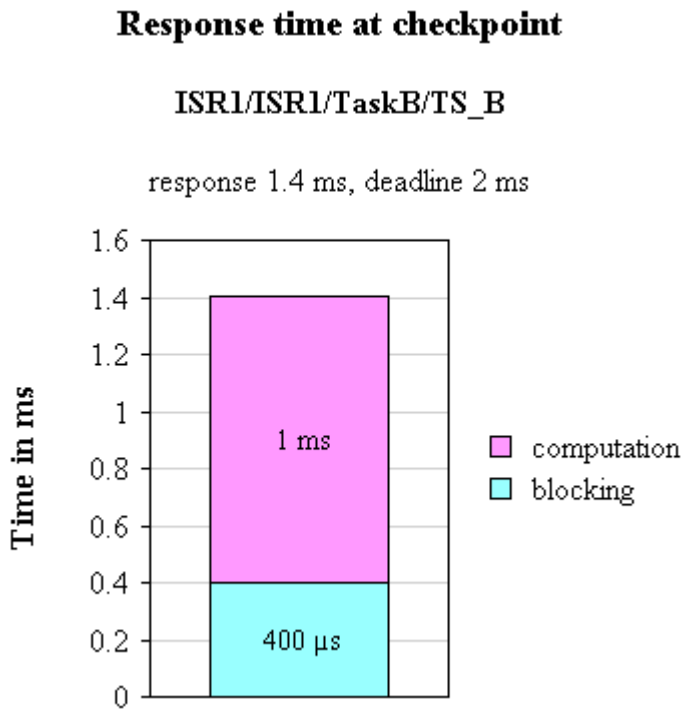
The transactions and checkpoints:



Quick look at green vote marks shows that both checkpoints are met.

Nothing changed in the checkpoint *Counter1/Alarm1/TaskA/TS_A*. Indeed, the subtransaction *ISR1/ISR1* may preempt *TaskA* as soon as ISR1 arrives when *TaskA* executes at it normal priority (i.e. 10).

However, the response time of checkpoint *ISR1/ISR1/TaskB/TS_B* changed:



The response time of the **high-priority subtransaction** is delayed by 400 us due to *blocking*. That happens because while *TaskA* executes the critical section, it boost the priority higher than *TaskB*. Therefore, *TaskB* can not start execution - in the worst case, for the duration of the access of *TaskA* to the critical section guarded by *Res1*. As soon as *TaskA* decreases it priority to normal value, the *TaskB* starts execution, actually preempting *TaskA*.

Click blocking bar brings the table that precisely shows what causes the blocking:

Section	Time	Priority	Type	Threshold
Counter1/Alarm1/OS/SA_GET_RESOURCE	0	OS		
Counter1/Alarm1/TaskA/TS_A_2	400 µs	20	activate	20
Counter1/Alarm1/OS/SA_RELEASE_RESOURCE...	0	OS		
Total blocking	400 µs			

In this case blocking is caused by the tasks section *TS_A_2* of *TaskA* that is executed at high priority 20, thus delaying start of *TaskB* for 400 us.

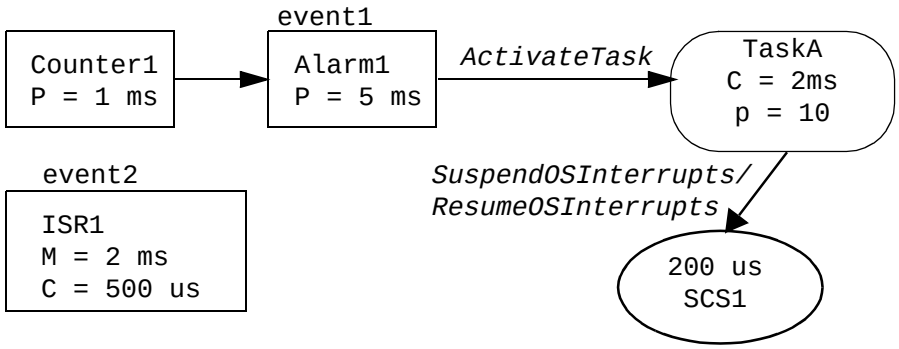
Blocking of the task or the ISR may occur only once - before start of the task or the ISR execution. Preemption of the task or the ISR may occur many times.

Periodic Task, an ISR, and Short Critical Section

The example is aimed to show the blocking of ISR caused by suspending interrupts in shorts critical section (SCS).

Event2 is fully processed by *ISR1*. Event1 is processed by *TaskA*, that needs to access hardware in order to generate response. Hardware module access can not be interrupted, and it is considered as short critical section, protected by means of suspending OS interrupts. *TaskA* needs 200 us to

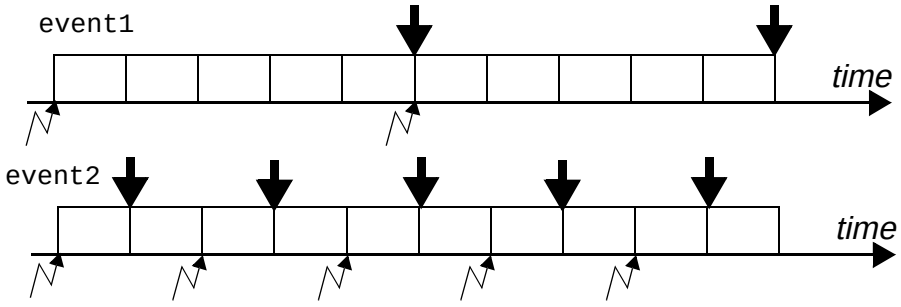
access the short critical section *SCS1*. The scheme of the application is shown below:



The *ISR1* has minimal interarrival time 2 milliseconds (ms), and takes 500 microseconds (us) of computational time to generate response before deadline 1 ms. *ISR1* generates response immediately before termination.

Event1 has the same timing requirements as described in **Periodic Task, an ISR and the Resource**.

Step 1. Timing requirements of applications events are presented on the picture:



Event1 arrives every 5 ms, and should be processed before deadline that is 5 ms from the arrival of the event. *Event2* arrives every 2 ms, and should be processed before deadline that is 1 ms from the arrival of the event.

Step2. Application has two transactions:

- *Counter1* that has period 1 ms
- *ISR1* that has minimal interarrival time 2 ms

Step3. Application subtransactions:

- Transaction *Counter1* has single subtransaction *Alarm1* (full name *Counter1/Alarm1*)
- Transaction *ISR1* has single subtransaction *ISR1* (full name *ISR1/ISR1*).

Step 4. The execution graphs:

- subtransaction *Counter1/Alarm1: TaskA* (more precisely, there are task sections *TS_A_1*, *TS_A_2*, that delimit an access to resource *SCS1*, and section *TS_A*). Task section *TS_A_1* is an initial section of *TaskA*
- subtransaction *ISR1/ISR1: ISR1* (more precisely, single ISR section *IS_1*)

Step 5. There is only one execution scenario for each subtransactions, that is the same as execution graph.

Step 6. Tasks and ISR sections are as following:

- *TaskA/TS_A_1*, that ends with an action *SuspendOSInterrupts*. Worst-case execution time is 800 us
- *TaskA/TS_A_2*, that ends with an action *ResumeOSInterrupts*. Worst-case execution time is 200 us. This section is actually critical section *SCS1*
- *TaskA/TS_A*, that ends with an action *TerminateTask*. Worst-case execution time of the section is 1 ms
- *ISR1/IS_1*, that ends with an action *LeaveISR*. Worst-case execution time of the section is 500 us

Step 7. The checkpoints are as following:

- Checkpoint is at the end of *TaskA*, when the task generates response and terminates. Full name of checkpoint is *Counter1/Alarm1/TaskA/TS_A*. The deadline is 5 ms
- Checkpoint is at the end of *ISR1*, when the ISR generates response and terminates. Full name of the checkpoint is *ISR1/ISR1/ISR1/IS_1_1*. The deadline is 1 ms

Step 8.OIL file for the example is presented below. The DS-Design Tool specific parameters are in **bold**.

```
// PeriodicTaskISRAndShortCS.oil

OIL_VERSION = "2.3";
#include "C:\metrowerks\osek\ostmpc\bin\ost22mpc.oil"
```

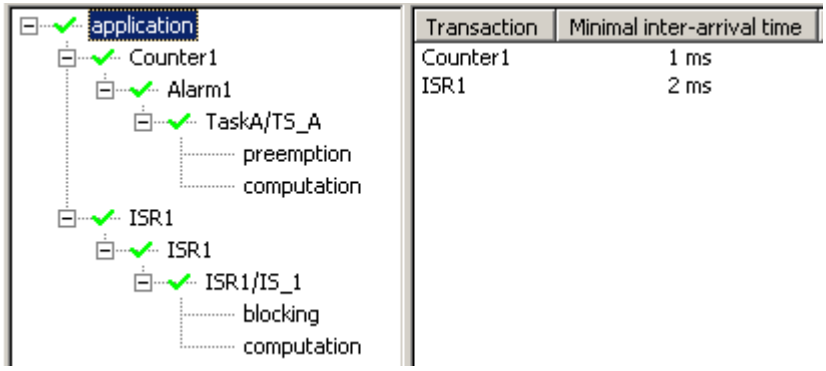
```
CPU cpu1
{
    APPMODE app1;
    OS os1
    {
      TargetMCU = MPC555;
      STATUS = STANDARD;
      ResourceScheduler = FALSE;
      STARTUPHOOK = FALSE;SHUTDOWNHOOK = FALSE;
      PRETASKHOOK = FALSE;POSTTASKHOOK = FALSE;ERRORHOOK = FALSE;
      USEGETSERVICEID = FALSE; USEPARAMETERACCESS = FALSE;

      saTimeUnit = us;
      saMeasureClockFrequency = 4000;
    };

    TASK TaskA
    {
      PRIORITY = 10;
      SCHEDULE = FULL;
      AUTOSTART = FALSE;
      ACTIVATION = 1;

      saInitial = "TS_A_1";
      saTaskSection = SET
      {
          ID = "TS_A_1";
          WCET = 800;
          ACTION = SuspendOSInterrupts { NEXT = "TS_A_2"; };
      };
      saTaskSection = SET
      {
          ID = "TS_A_2";
          WCET = 200;
          ACTION = ResumeOSInterrupts { NEXT = "TS_A"; };
      };
      saTaskSection = SET { ID = "TS_A";
          WCET = 1000;
          DEADLINE = SET { VALUE = 5000; };
          ACTION = TerminateTask;
      };
    };

    COUNTER Counter1
    {
      MAXALLOWEDVALUE = 0xffffffff; MINCYCLE = 0; TICKSPERBASE = 1;
      saPeriod = 1000;
    };

    ALARM Alarm1
    {
      COUNTER = Counter1;
      ACTION = ACTIVATETASK { TASK = TaskA; };
```

```
      AUTOSTART = FALSE;
      saCyclic = TRUE { ALARMTIME = 0; CYCLETIME = 5; };
    };

    ISR ISR1
    {
      CATEGORY = 2;
      PRIORITY = 10;

      saMinIATime = 2000;

      saISRSection = SET { ID = "IS_1";
          WCET = 500;
          DEADLINE = SET { VALUE = 1000; };
          ACTION = LeaveISR;
      };
    };
};
```

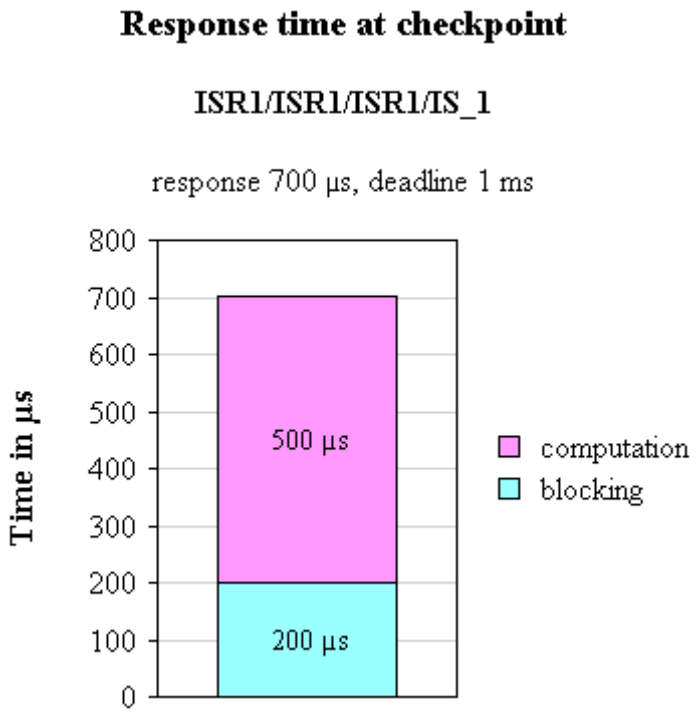
Step 9. Run DS-Design Tool.

Step 10. The result of the analysis are explained below.

The transactions and checkpoints:



Both deadlines are met, though response time in interrupt service routine is delayed for 200 us:



Click on blocking bar explains the cause of blocking:

Section	Time	Priority	Type	Thresh...
Counter1/Alarm1/OS/SA_SUSPEND_OS_INT...	0	OS		
Counter1/Alarm1/TaskA/TS_A_2	200 µs	10	activate	10
Counter1/Alarm1/OS/SA_RESUME_OS_INTE...	0	OS		
Total blocking	200 µs			

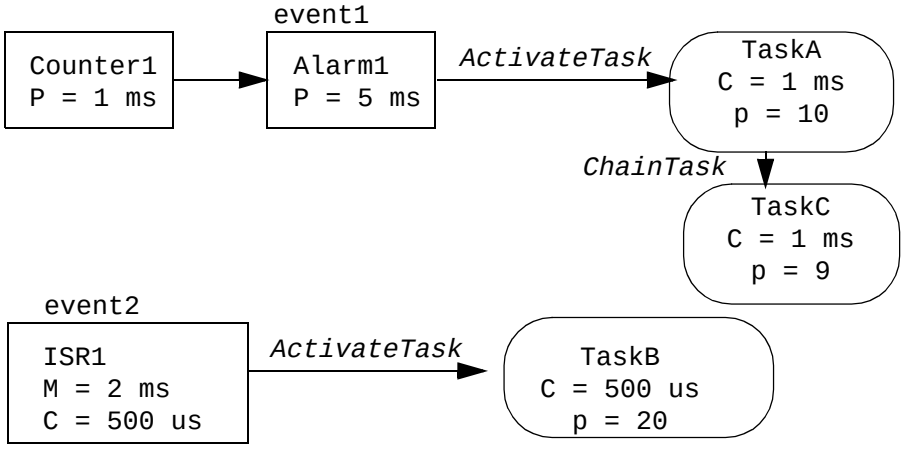
The *TaskA* short critical section *TS_A_2* disables interrupts thus delaying release of an *ISR1*.

Periodic Task, an ISR, and GoTo actions

The example the a modification of the example in subsection **Periodic Task and an ISR**, therefore here only additional details are described.

The example aims to demonstrate the use of *GoTo* action to analyze conditional operators, and multiple deadlines in a subtransaction. It also shows missing deadlines.

Event1 is processed by the chain of tasks *TaskA* and *TaskC*. *TaskA* has a conditional operator (explained further), and computational time of each branch differs. There are two responses generated to event1 - the first by *TaskA*, and the second by *TaskC*.

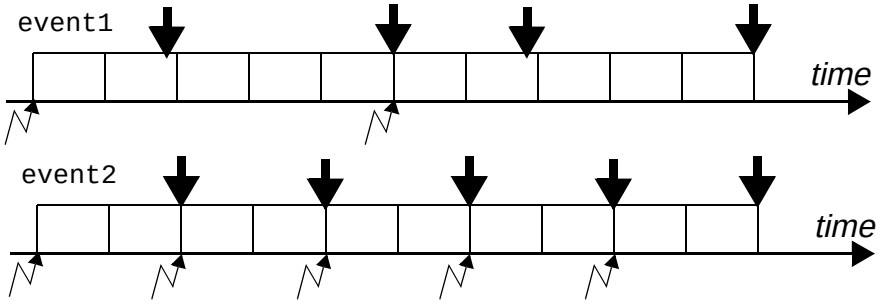


Task*TaskA* has the following structure. The task checks the value of variable *x*, and executes different branches for positive value of *x* and for non-positive value.The numbers in comments are worst-case execution times of the corresponding sections:

```
TASK(TaskA)
{
 TS_A_1

if(x>0) /*200 us*/
{
 TS_A_2
 f1(); /*300 us*/
}
 TS_A_3
else
{
 f2(); /*100 us*/
}
 TS_A
ChainTask(TaskC); /*500 us*/
}
```

Note that total execution time of *TaskA* is 1000 us if value of variable *x* is positive, and 800 us otherwise.

Step 1. Timing requirements of applications events are presented on the picture:



Event1 arrives every 5 ms, and has two deadlines. The first is 1.9 ms, while the second is 5 ms (both times from the arrival of the event). *Event2* arrives every 2 ms, and should be processed before deadline, that is 2 ms from the arrival of the event.

Step2. Application has two transactions:

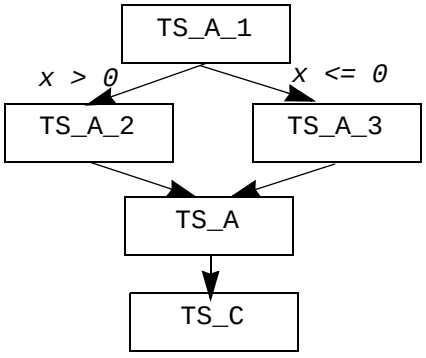
- *Counter1* that has period 1 ms
- *ISR1* that has minimal interarrival time 2 ms

Step3. Application subtransactions:

- Transaction *Counter1* has single subtransaction *Alarm1* (full name *Counter1/Alarm1*)
- Transaction *ISR1* has single subtransaction *ISR1* (full name *ISR1/ISR1*).

Step 4. The execution graphs:

- subtransaction *Counter1/Alarm1*:



- subtransaction *ISR1/ISR1*: *ISR1* and *TaskB* (more precisely, two ISR sections of ISR1, identified as *IS_1_1* and *IS_1_2*, and single task section *TS_B*). ISR section *IS_1_1* is an initial section of *ISR1*

Step 5. The following execution scenarios exists:

Execution scenario for subtransaction *Counter1/Alarm1*:

- *TS_A_1*, *TS_A_2*, *TS_A*, *TS_C* (positive value of *x*)
- *TS_A_1*, *TS_A_3*, *TS_A*, *TS_C* (non positive value of *x*)

Execution scenario for subtransaction *ISR1/ISR1*:

- *IS_1_1*, *IS_1_2*, *TS_B*

Step 6. Tasks and ISR sections are as following:

- *TaskA/TS_A_1*, that ends with two *GoTo* actions. First *GoTo* jumps to *TS_A_2*. Second *GoTo* jumps to *TS_A_3*. The worst-case execution time of the section is 200 us
- *TaskA/TS_A_2*, that ends with *GoTo* actions to *TS_A*. Worst-case execution time of the section is 300 us. This is the branch for positive value of *x*
- *TaskA/TS_A_3*, that ends with *GoTo* actions to *TS_A*. Worst-case execution time of the section is 100 us. This is the branch for non-positive value of *x*
- *TaskA/TS_A*, that ends with action *ChainTask*. Worst-case execution time of the section is 500 us
- *TaskC/TS_C*, that ends with action *TerminateTask*. Worst-case execution time of the section is 1000 us
- *ISR1/IS_1_1*, that ends with action *ActivateTask*, and is followed by the section *IS_1_2*. Worst-case execution time of the section is 300 us
- *ISR1/IS_1_2*, that ends with action *LeaveISR*. Worst-case execution time of the section is 200 us
- *TaskB/TS_B*, that ends with action *TerminateTask*. Worst-case execution time of the section is 500 us

Step 7. The checkpoints are as following:

- Checkpoint is at the end of *TaskA*, when the task generates the first response of the subtransaction, and chains to *TaskC*. Full name of the checkpoint is *Counter1/Alarm1/TaskA/TS_A*. The deadline is 1.9 ms

- Checkpoint is at the end of *TaskC*, when the task generates the second response of the subtransaction, and terminates. Full name of the checkpoint is *Counter1/Alarm1/TaskA/TS_C*. The deadline is 5 ms
- Checkpoint is at the end of *TaskB*, when the task generates response and terminates. Full name of checkpoint is *ISR1/ISR1/TaskB/TS_B*. The deadline is 2 ms

Step 8.OIL file for the example is presented below. The DS-Design Tool specific parameters are in **bold**.

```
// PeriodicTaskISRAndGoto.oil

OIL_VERSION = "2.3";
#include "C:\metrowerks\osek\ostmpc\bin\ost22mpc.oil"

CPU cpu1
{
    APPMODE app1;
    OS os1
    {
      TargetMCU = MPC555;
      STATUS = STANDARD;
      ResourceScheduler = FALSE;
      STARTUPHOOK = FALSE;SHUTDOWNHOOK = FALSE;
      PRETASKHOOK = FALSE;POSTTASKHOOK = FALSE;ERRORHOOK = FALSE;
      USEGETSERVICEID = FALSE; USEPARAMETERACCESS = FALSE;

      saTimeUnit = us;
      saMeasureClockFrequency = 4000;
    };

    TASK TaskA
    {
      PRIORITY = 10;
      SCHEDULE = FULL;
      AUTOSTART = FALSE;
      ACTIVATION = 1;

      saInitial="TS_A_1";
      saTaskSection = SET { ID = "TS_A_1";
          WCET = 200;
          ACTION = GoTo { NEXT = "TS_A_2"; };
          ACTION = GoTo { NEXT = "TS_A_3"; };
      };
      saTaskSection = SET { ID = "TS_A_2";
          WCET = 300;
          ACTION = GoTo { NEXT = "TS_A"; };
      };
      saTaskSection = SET { ID = "TS_A_3";
          WCET = 100;
```

```
      ACTION = GoTo { NEXT = "TS_A"; };
  };
  saTaskSection = SET { ID = "TS_A";
      WCET = 500;
      DEADLINE = SET { VALUE = 1900; };
      ACTION = ChainTask { TASK = TaskC; };
  };
};

TASK TaskC
{
  PRIORITY = 9;
  SCHEDULE = FULL;
  AUTOSTART = FALSE;
  ACTIVATION = 1;

  saTaskSection = SET { ID = "TS_C";
      WCET = 1000;
      DEADLINE = SET { VALUE = 5000; };
      ACTION = TerminateTask;
  };
};

COUNTER Counter1
{
  MAXALLOWEDVALUE = 0xffffffff; MINCYCLE = 0; TICKSPERBASE = 1;
  saPeriod = 1000;
};

ALARM Alarm1
{
  COUNTER = Counter1;
  ACTION = ACTIVATETASK { TASK = TaskA; };
  AUTOSTART = FALSE;
  saCyclic = TRUE { ALARMTIME = 0; CYCLETIME = 5; };
};

TASK TaskB
{
  PRIORITY = 20;
  SCHEDULE = FULL;
  AUTOSTART = FALSE;
  ACTIVATION = 1;

  saTaskSection = SET { ID = "TS_B";
      WCET = 500;
      DEADLINE = SET { VALUE = 2000; };
      ACTION = TerminateTask;
  };
};

ISR ISR1
{
  CATEGORY = 2;
```

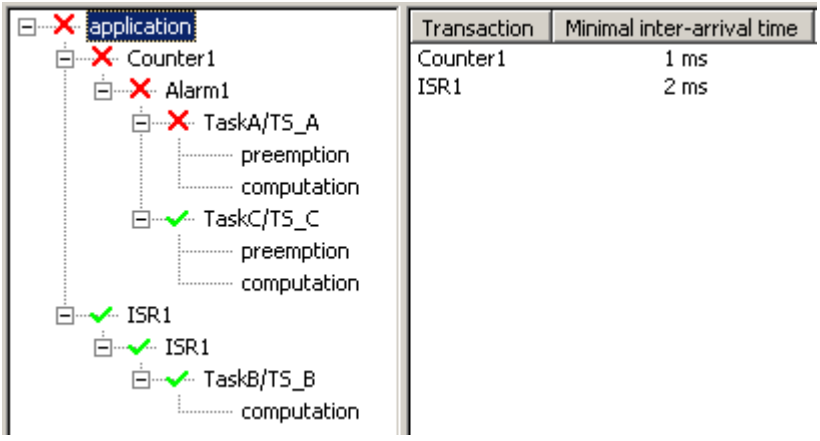
```
        PRIORITY = 10;

        saInitial="IS_1_1";
        saMinIATime = 2000;

        saISRSection = SET { ID = "IS_1_1";
            WCET = 300;
            ACTION = ActivateTask { NEXT = "IS_1_2"; TASK = TaskB; };
        };

        saISRSection = SET { ID = "IS_1_2";
            WCET = 200;
            ACTION = LeaveISR;
        };
    };
};
```

Step 9. Run DS-Design Tool.

Step 10. The result of the analysis are explained below.

If there are several paths to the checkpoint DS-Design Tool considers them during analysis and displays the path with maximal response time.

The transactions and checkpoints of the application are as following:



DS-Design Tool has considered the following and created two checkpoints for subtransaction *Counter1/Alarm1* (3 and 4 selected):

1. *Counter1/Alarm1/TaskA/TS_A*:
   Response time 1.8 ms, Deadline 1.9 ms, deadline met

Computation time is 800 us (*TS_A_1+TS_A_3+TS_A*)
Preemption time is 1 ms (*ISR1+TaskB*)

2. *Counter1/Alarm1/TaskC/TS_C*:
   Response time 3.8 ms, Deadline 5 ms, deadline met
   Computation time is 1.8 us (*TS_A_1+TS_A_3+TS_A+TS_C*)
   Preemption time is 2 ms (two times of *ISR1+TaskB*)
3. *Counter1/Alarm1/TaskA/TS_A*:
   Response time 2 ms, Deadline 1.9 ms, deadline **missed**
   Computation time is 1000 us (*TS_A_1+TS_A_2+TS_A*)
   Preemption time is 1 ms (*ISR1+TaskB*)
4. *Counter1/Alarm1/TaskC/TS_C*:
   Response time 4 ms, Deadline 5 ms, deadline met
   Computation time is 2 us (*TS_A_1+TS_A_2+TS_A+TS_C*)
   Preemption time is 2 ms (two times of *ISR1+TaskB*)

The checkpoints show that when subtransaction is executed the scenario for positive value of *x*, the first deadline is missed, because *TaskA* takes 1000 us to compute response for positive value of *x*. For non-positive value *TaskA* takes only 800 us, and first deadline is met. All other deadlines are met regarding the execution scenario.

If the application utilization is higher than 100%, then DS-Design Tool does not calculate response time for missed deadlines. In this case DS-Design Tool shows utilization diagram for these deadlines.

Periodic Task, an ISR, and Paths

The example is modification of the example in subsection **Periodic Task, an ISR, and GoTo actions**, therefore here only additional details are described.

The example aims to demonstrate the use of *PATH* parameter to analyze dependent conditional operators.

The scheme of an application is the same. The only difference is that *TaskC* also uses conditional operator checking the same condition as *TaskA* does.

Therefore, task *TaskC* has the following structure. The task checks the value of variable *x*, and executes different branches for positive value of *x*

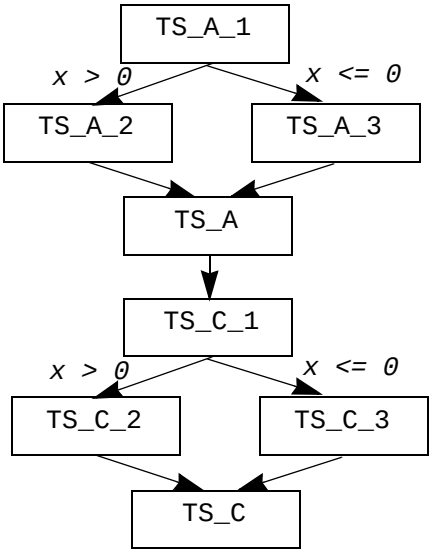
and for non-positive value.The numbers in comments are worst-case execution times of the corresponding sections:

```
TASK(TaskC)
{
TS_C_1

if(x>0) /*200 us*/
TS_C_2  {
         q1(); /*300 us*/
        }
TS_C_3  else
        {
         q2(); /*100 us*/
        }
TS_C    TerminateTask(); /*500 us*/
        }
```

Note that total execution time of *TaskC* is 1000 us if the value of variable *x* is positive, and 800 us otherwise. The numbers are exactly the same as for *TaskA* - this is only to simplify explanation.

Step 1, **Step2**, and **Step3** are not changed.

Step 4. The execution graphs for subtransaction *Counter1/Alarm1* becomes more complex:

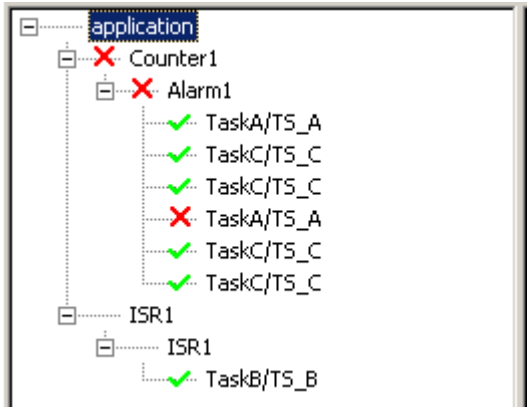


Note that execution graph potentially comprises **four** execution scenarios, if the conditions **are not taken into consideration**. Indeed, the following scenarios **could** exist:

1. *TS_A_1*, *TS_A_2*, *TS_A*, *TS_C_1*, *TS_C_2*, *TS_C*
2. *TS_A_1*, *TS_A_2*, *TS_A*, *TS_C_1*, *TS_C_3*, *TS_C*
3. *TS_A_1*, *TS_A_3*, *TS_A*, *TS_C_1*, *TS_C_2*, *TS_C*
4. *TS_A_1*, *TS_A_3*, *TS_A*, *TS_C_1*, *TS_C_3*, *TS_C*

Clearly, scenario 2 is wrong, because sections *TS_A_2* and *TS_C_3* are for vise versa condition. Also scenario 4 is wrong, as sections *TS_A_3* and *TS_C_2* contradict each other.

The picture below shows what **would** happen if wrong scenarios would not be removed:



DS-Design Tool counts **all** scenarios, and builds 4 checkpoints for second deadline, as it believes four routes go there (two forks).

Obviously, the wrong scenarios must be avoided. In this case the paths mechanism helps.

Step 5. Valid execution scenarios for subtransaction *Counter1/Alarm1*.

Valid scenario 1 is for positive value of *x*. Therefore, task sections of this scenario belong to the path that is named *Xgt0* (*x great than zero*). Valid scenario 3 is for non positive value of *x*. Therefore, task sections of this scenario belong to path that is named *Xle0* (*x less or equal zero*). These path names are used in action parameters of task sections to teach DS-Design Tool valid scenarios:

- *TS_A_1*, *TS_A_2*, *TS_A*, *TS_C_1*, *TS_C_2*, *TS_C* (path *Xgt0*)
- *TS_A_1*, *TS_A_3*, *TS_A*, *TS_C_1*, *TS_C_3*, *TS_C* (path *Xle0*)

Step 6. Task sections of the subtransaction *Counter1/Alarm1* are as following:

- *TaskA/TS_A_1*, that ends with two *GoTo* actions. First *GoTo* jumps to *TS_A_2* for path *Xgt0*. Second *GoTo* jumps to *TS_A_3* for path *Xle0*. Worst-case execution time of the section is 200 us
- *TaskA/TS_A_2*, that ends with *GoTo* actions to *TS_A* for path *Xgt0*. Worst-case execution time of the section is 300 us. This is the branch for positive value of *x*
- *TaskA/TS_A_3*, that ends with *GoTo* actions to *TS_A* for path *Xle0*. Worst-case execution time of the section is 100 us. This is the branch for non-positive value of *x*
- *TaskA/TS_A*, that ends with an action *ChainTask*. Worst-case execution time of the section is 500 us
- *TaskC/TS_C_1*, that ends with two *GoTo* actions. First *GoTo* jumps to *TS_C_2* for path *Xgt0*. Second *GoTo* jumps to *TS_C_3* for path *Xle0*. Worst-case execution time of the section is 200 us
- *TaskC/TS_C_2*, that ends with *GoTo* actions to *TS_C* for path *Xgt0*. Worst-case execution time of the section is 300 us. This is the branch for positive value of *x*
- *TaskC/TS_C_3*, that ends with *GoTo* actions to *TS_C* for path *Xle0*. Worst-case execution time of the section is 100 us. This is the branch for non-positive value of *x*
- *TaskC/TS_C*, that ends with an action *TerminateTask*. The worst-case execution time of the section is 500 us

Step 7. The checkpoints are not changed.

Step 8.OIL file for the example is presented below. The DS-Design Tool specific parameters are in **bold**.

```
// PeriodicTaskISRAndPaths.oil

OIL_VERSION = "2.3";
#include "C:\metrowerks\osek\ostmpc\bin\ost22mpc.oil"

CPU cpu1
{
    APPMODE app1;
    OS os1
```

```
{
  TargetMCU = MPC555;
  STATUS = STANDARD;
  ResourceScheduler = FALSE;
  STARTUPHOOK = FALSE;SHUTDOWNHOOK = FALSE;
  PRETASKHOOK = FALSE;POSTTASKHOOK = FALSE;ERRORHOOK = FALSE;
  USEGETSERVICEID = FALSE; USEPARAMETERACCESS = FALSE;

  saTimeUnit = us;
  saMeasureClockFrequency = 4000;
};

TASK TaskA
{
  PRIORITY = 10;
  SCHEDULE = FULL;
  AUTOSTART = FALSE;
  ACTIVATION = 1;

  saInitial="TS_A_1";
  saTaskSection = SET { ID = "TS_A_1";
      WCET = 200;
      ACTION = GoTo { NEXT = "TS_A_2"; PATH = "Xgt0"; };
      ACTION = GoTo { NEXT = "TS_A_3"; PATH = "Xle0"; };
  };
  saTaskSection = SET { ID = "TS_A_2";
      WCET = 300;
      ACTION = GoTo { NEXT = "TS_A"; PATH = "Xgt0"; };
  };
  saTaskSection = SET { ID = "TS_A_3";
      WCET = 100;
      ACTION = GoTo { NEXT = "TS_A"; PATH = "Xle0"; };
  };
  saTaskSection = SET { ID = "TS_A";
      WCET = 500;
      DEADLINE = SET { VALUE = 1900; };
      ACTION = ChainTask { TASK = TaskC; };
  };
};

TASK TaskC
{
  PRIORITY = 9;
  SCHEDULE = FULL;
  AUTOSTART = FALSE;
  ACTIVATION = 1;

  saInitial="TS_C_1";
  saTaskSection = SET { ID = "TS_C_1";
      WCET = 200;
      ACTION = GoTo { NEXT = "TS_C_2"; PATH = "Xgt0"; };
      ACTION = GoTo { NEXT = "TS_C_3"; PATH = "Xle0"; };
  };
  saTaskSection = SET { ID = "TS_C_2";
```

```
      WCET = 300;
      ACTION = GoTo { NEXT = "TS_C"; PATH = "Xgt0"; };
  };
  saTaskSection = SET { ID = "TS_C_3";
      WCET = 100;
      ACTION = GoTo { NEXT = "TS_C"; PATH = "Xle0"; };
  };
  saTaskSection = SET { ID = "TS_C";
      WCET = 500;
      DEADLINE = SET { VALUE = 5000; };
      ACTION = TerminateTask;
  };
};

COUNTER Counter1
{
  MAXALLOWEDVALUE = 0xffffffff; MINCYCLE = 0; TICKSPERBASE = 1;
  saPeriod = 1000;
};

ALARM Alarm1
{
  COUNTER = Counter1;
  ACTION = ACTIVATETASK { TASK = TaskA; };
  AUTOSTART = FALSE;
  saCyclic = TRUE { ALARMTIME = 0; CYCLETIME = 5; };
};

TASK TaskB
{
  PRIORITY = 20;
  SCHEDULE = FULL;
  AUTOSTART = FALSE;
  ACTIVATION = 1;

  saTaskSection = SET { ID = "TS_B";
      WCET = 500;
      DEADLINE = SET { VALUE = 2000; };
      ACTION = TerminateTask;
  };
};

ISR ISR1
{
  CATEGORY = 2;
  PRIORITY = 10;

  saInitial="IS_1_1";
  saMinIATime = 2000;

  saISRSection = SET { ID = "IS_1_1";
      WCET = 300;
      ACTION = ActivateTask { NEXT = "IS_1_2"; TASK = TaskB; };
  };
```

```
        saISRSection = SET { ID = "IS_1_2";
            WCET = 200;
            ACTION = LeaveISR;
        };
    };
};
```

Step 9. Run DS-Design Tool.

Step 10. The result of the analysis are explained below.

The transactions and checkpoints of the application are as following:

Note that DS-Design Tool has considered **two** valid scenarios, and it is the paths mechanism that helped DS-Design Tool to get rid of wrong scenarios.

DS-Design Tool has created two checkpoints for subtransaction *Counter1/Alarm1* (1 and 2 selected).

1. *Counter1/Alarm1/TaskA/TS_A*:
   Response time 2 ms, Deadline 1.9 ms, deadline **missed**
   Computation time is 1000 us (*TS_A_1+TS_A_2+TS_A*)
   Preemption time is 1 ms (*ISR1+TaskB*)
2. *Counter1/Alarm1/TaskC/TS_C*:
   Response time 4 ms, Deadline 5 ms, deadline met
   Computation time is 2 us
   (*TS_A_1+TS_A_2+TS_A+TS_C_1+TS_C_2+TS_C*)
   Preemption time is 2 ms (two times of *ISR1+TaskB*)

3. *Counter1/Alarm1/TaskA/TS_A*:
   Response time 1.8 ms, Deadline 1.9 ms, deadline met
   Computation time is 800 us (*TS_A_1+TS_A_3+TS_A*)
   Preemption time is 1 ms (*ISR1+TaskB*)
4. *Counter1/Alarm1/TaskC/TS_C*:
   Response time 3.6 ms, Deadline 5 ms, deadline met
   Computation time is 1.6 us
   (*TS_A_1+TS_A_3+TS_A+TS_C_1+TS_C_3+TS_C*)
   Preemption time is 2 ms (two times of *ISR1+TaskB*)

Therefore, *PATH* parameter in *ACTION* of the task and ISR sections allows effectively remove excessive and invalid scenarios.

Tuning an Application

DS-Design Tool is an analysis tool that helps to understand timing behavior of an application. In case the application behavior is not proper, the application designer needs to adjust the application in order to make it scheduled.

This subsection contains number of recommendations and examples that may be used as a guidelines. This subsection contains references to the examples in **Applying Computational Model** where appropriate.

WARNING! The DS-Design Tool is not a "silver bullet" meaning it can not solve all schedulability problems.

General Recommendations

DS-Design Tool results should be explored for missed deadlines first.

Deadline misses may be caused by the over-utilization of processor. Therefore, utilization chart should be explored to check if utilization is below 100%.

If utilization is below 100%, then response times of all missed deadlines should be explored.

In general case, the response time chart includes computation time, preemption time, and blocking time.

The computation time is a sum of worst-case execution times of all tasks and ISRs that the execution scenario comprises. DS-Design Tool shows breakdown of the response time, and correctness of the scenario and numbers, which should be checked to be sure that the response is generated by proper scenario.

The preemption time shows the sum of execution times of all entities that could preempt the subtransaction from the processor, because these entities have **higher** priority. If preemption time causes the delay of the response, the entities that contribute into preemption should be explored. Some of the entities may have unreasonably high priorities, which may be decreased without making any harm to other checkpoints. To understand which entities may be decreased in the priority, the met deadlines could be explored. If the response time is well below deadline, then the priority of the entities that contribute into the response should be revised and possibly decreased.

The blocking time shows the contribution of the task or ISR section of the entity of the **lower** priority. The blocking is caused by temporary increase of priority due to access to the critical section (as OSEKturbo uses OSEK Priority Ceiling Protocol for resource management). Alternatively, the cause of blocking may be the non-preemptivness of the task. The subsections Restructuring Critical Sections and Changing Scheduling Policy below explain the possible methods of blocking decrease.

Specific Recommendations

Next subsections describe in details several methods that can be used to adjust timing behavior of the application and improve scheduling. Here is the list of these methods:

1. Rethinking Hardness of Deadlines
2. Increasing Speed of Processor
3. Using Timescale for Periodic Tasks
4. Making Periods Harmonic
5. Restructuring subtransactions

6. Restructuring Critical Sections
7. Changing Scheduling Policy
8. Employing OSEKturbo Internal Resources
9. Using Paths

Rethinking Hardness of Deadlines

In the example **Periodic Task, an ISR, and GoTo actions** the first deadline of subtransaction *Counter1/Alarm1*, generated at the end of *TaskA*, is not met. Deadline is 1.9 ms, while response is 2 ms.

The first idea is to relax the deadline. Will application really fail, if the deadline is increased by 100 us? If not, it makes sense to do that, and get the application scheduled.

The second idea is to rethink hardness of the deadline. Does the application fail if deadline is not met? If not, the deadline may be considered as the soft deadline, and therefore application may miss it[1].

Increasing Speed of Processor

The processor speed basically impacts execution times of tasks, ISRs, and operating system services. Therefore, speed increase can make execution times shorter while deadlines and timers (counters) and alarms values remain unchanged.

DS-Design Tool uses parameter *saClockFrequency* to specify the current oscillator frequency. Therefore, increase of the value of *saClockFrequency* automatically decreases execution times such as WCETs and operating system overheads. In order to specify that measurement of those execution times was done using different clock frequency, the parameter *saMeasureClockFrequency* should be set.

However, certain processors (e.g. MGT5x00) are complicated enough, and increasing of oscillator frequency does not lead to proportional decreasing of execution times. Thus special invalidation flag *saInvalidate* controls timing adjustment. If this flag is set to TRUE DS-Design Tool will warn

1. The soft deadline should be inspected by the application developer.

user about potential invalid anaylsis results in case when current oscillator frequency differs from the one at which execution times were measured.

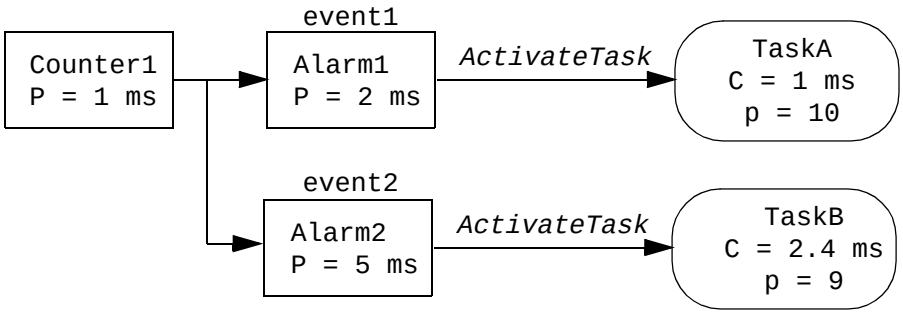
Using Timescale for Periodic Tasks

The OSEKturbo timescale is an implementation of OSEK/VDX OS static alarms. Timescale allows to build the schedule of tasks execution, and it suits well the periodic events processing.

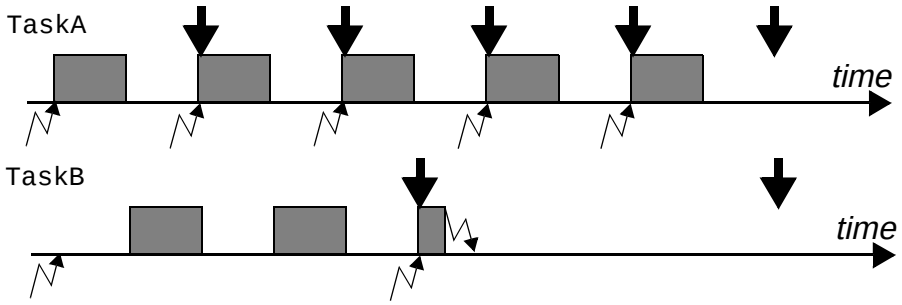
As tasks in timescale are executed with a user-defined offset, there is dependency between the releases of tasks. The dependencies decrease the pessimism of the analysis as the tasks are not released simultaneously.

Therefore, for periodic tasks timescale is often very good solution that improves scheduling dramatically.

For example, the following application with two periodic tasks is not scheduled if the deadline of each task is equal to the period:



The timing diagram shows how the tasks are executed on the processor (execution of the *TaskB* is not shown after the first miss of deadline)[1]:

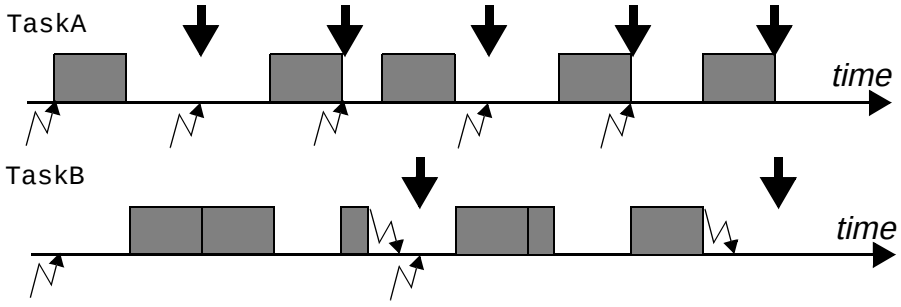


DS-Design Tool considers releases of both tasks as independent events, and therefore assumes that may arrive simultaneously. As *TaskA* has

higher priority, it preempts the execution of *TaskB*, and therefore *TaskB* misses deadline. The small filled box in the bottom time line shows the 400 us execution time of *TaskB*, which is done behind the deadline.

Indeed, DS-Design Tool shows that response time at checkpoint *Counter1/Alarm2/TaskB/TS_B* is 5.4 ms, and the contribution of the preemption is 3 ms (3 filled boxes in the top time line on the picture above).

To employ timescale, the task schedule of the application should be built. For example, it may look like the following one[1]:



OIL file for the example is presented below. The DS-Design Tool specific parameters are in **bold**. The timescale specific attributes are in *italic*.

```
// Timescale-Good.oil

OIL_VERSION = "2.3";
#include "ost22mpc.oil"

CPU cpu1
{
    APPMODE app1;
    OS os1
    {
      TargetMCU = MPC555
      {
        SysTimer = HWCOUNTER
        {
         COUNTER = Counter1;
         ISRPRIORITY = 14;
```

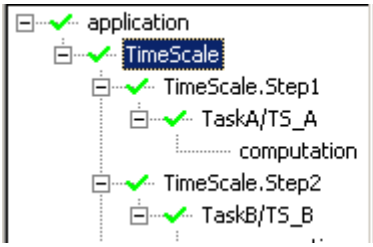
1. Responses of *TaskA* are not shown as they obviously are generated at the end of each filled box.

1. Responses of *TaskA* are not shown as they obviously are generated at the end of each filled box.

```
      TimerHardware = TB0 { Prescaler = OS { Value = 0; }; };
    };
  };

  STATUS = STANDARD;
  ResourceScheduler = FALSE;
  STARTUPHOOK = FALSE;SHUTDOWNHOOK = FALSE;
  PRETASKHOOK = FALSE;POSTTASKHOOK = FALSE;ERRORHOOK = FALSE;
  USEGETSERVICEID = FALSE; USEPARAMETERACCESS = FALSE;

  saTimeUnit = us;
  saMeasureClockFrequency = 4000;

  TimeScale = TRUE
  {
  TimeUnit = us;
  ScalePeriod = 10000;
  Step = SET { StepNumber = 1; TASK = TaskA; StepTime = 1000; };
  Step = SET { StepNumber = 2; TASK = TaskB; StepTime = 2000; };
  Step = SET { StepNumber = 3; TASK = TaskA; StepTime = 1500; };
  Step = SET { StepNumber = 4; TASK = TaskA; StepTime = 1000; };
  Step = SET { StepNumber = 5; TASK = TaskB; StepTime = 1500; };
  Step = SET { StepNumber = 6; TASK = TaskA; StepTime = 2000; };
  Step = SET { StepNumber = 7; TASK = TaskA; StepTime = 1000; };
  };
};

TASK TaskA
{
  PRIORITY = 10;
  SCHEDULE = FULL;
  AUTOSTART = FALSE;
  ACTIVATION = 1;

  saTaskSection = SET { ID = "TS_A";
      WCET = 1000;
      DEADLINE = SET { VALUE = 2000; };
      ACTION = TerminateTask;
  };
};

TASK TaskB
{
  PRIORITY = 9;
  SCHEDULE = FULL;
  AUTOSTART = FALSE;
  ACTIVATION = 1;

  saTaskSection = SET { ID = "TS_B";
      WCET = 2400;
      DEADLINE = SET { VALUE = 5000; };
      ACTION = TerminateTask;
  };
};
```

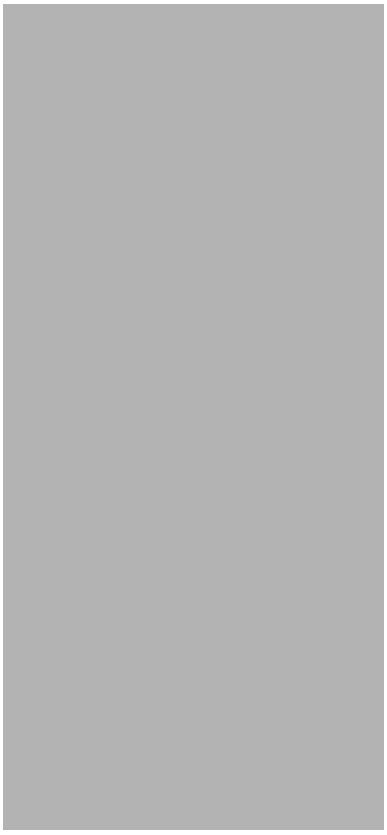
```
    COUNTER Counter1
    {
      MAXALLOWEDVALUE = 0xffffffff; MINCYCLE = 0; TICKSPERBASE = 1;
    };
};
```

DS-Design Tool reflects the timescale steps as subtransactions:

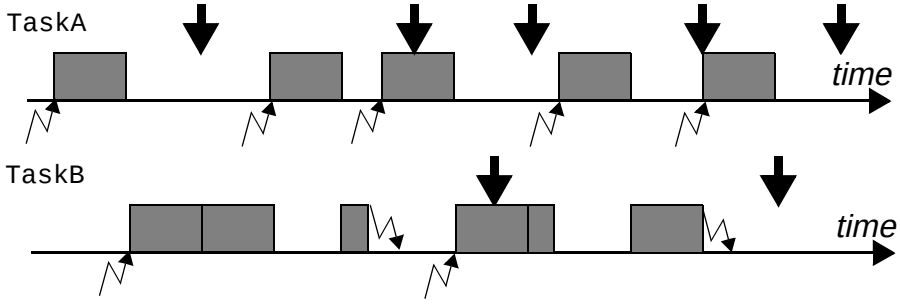


Subtransaction	Every	Offset	Jitter
TimeScale.Step1	1	0	0
TimeScale.Step2	1	1 ms	0
TimeScale.Step3	1	3 ms	0
TimeScale.Step4	1	4.5 ms	0
TimeScale.Step5	1	5.5 ms	0
TimeScale.Step6	1	7 ms	0
TimeScale.Step7	1	9 ms	0

Then, DS-Design Tool shows that all deadlines are met:



Note that DS-Design Tool considers the deadlines and response times starting the release time of subtransaction, so DS-Design Tool virtually uses timing diagram that looks like the following one[1]:



Making Periods Harmonic

It is theoretically proved that harmonic periods of events improve the scheduling of an application (see **[7]** in section "References,")

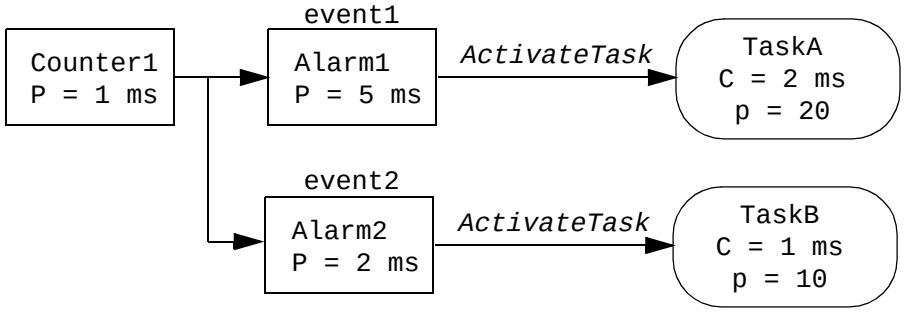
Restructuring subtransactions

Often the priorities of tasks and especially ISRs can not be easy changed though they not necessarily need high-priority for execution of the whole entity. If a high-priority task or an ISR delays the execution of tasks of lower priorities, then it makes sense to consider the splitting of the high-priority task and the ISR into two entities. The first one should be run at high priority, while the second may do the rest of work at lower priority level.

In the application two alarms are attached to one counter *Counter1* that ticks every 1 ms. *Alarm1* has cycle value 5 ms, and it activates high-

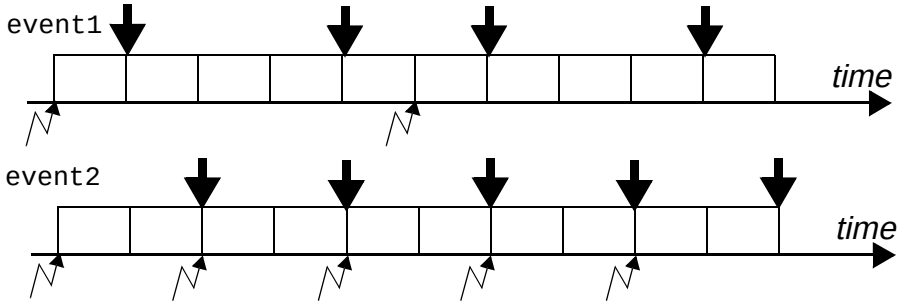
1. Responses of *TaskA* are not shown as they obviously are generated at the end of each filled box.

priority task *TaskA*. *Alarm2* has cycle value 2 ms, and it activates low-priority task *TaskB*:



TaskA generates two responses - first response requires 1 ms of computations and has deadline 1 ms, second response requires additional 1 ms of computations, and has deadline 4 ms. *TaskB* generate one response that requires 1 ms of computations and has deadline 2 ms.

The priority allocation is explained by the application deadlines:



As *TaskA* has critical first deadline, the task has high priority.

OIL file for the example is presented below. DS-Design Tool specific parameters are in **bold**. The objects and parameters that will be adjusted are in *italic*.

```
// TwoPeriodicTasks-Bad.oil

OIL_VERSION = "2.3";
#include "ost22mpc.oil"

CPU cpu1
{
    APPMODE app1;
    OS os1
```

```
{
  TargetMCU = MPC555;
  STATUS = STANDARD;
  ResourceScheduler = FALSE;
  STARTUPHOOK = FALSE;SHUTDOWNHOOK = FALSE;
  PRETASKHOOK = FALSE;POSTTASKHOOK = FALSE;ERRORHOOK = FALSE;
  USEGETSERVICEID = FALSE; USEPARAMETERACCESS = FALSE;

  saTimeUnit = us;
  saMeasureClockFrequency = 4000;
};

TASK TaskA
{
  PRIORITY = 20;
  SCHEDULE = FULL;
  AUTOSTART = FALSE;
  ACTIVATION = 1;

  saInitial = "TS_A_1";
  saTaskSection = SET { ID = "TS_A_1";
      WCET = 1000;
      DEADLINE = SET { VALUE = 1000; };
      ACTION = GoTo { NEXT = "TS_A"; };
  };
  saTaskSection = SET { ID = "TS_A";
      WCET = 1000;
      DEADLINE = SET { VALUE = 4000; };
      ACTION = TerminateTask;
  };
};

TASK TaskB
{
  PRIORITY = 10;
  SCHEDULE = FULL;
  AUTOSTART = FALSE;
  ACTIVATION = 1;

  saTaskSection = SET { ID = "TS_B";
      WCET = 1000;
      DEADLINE = SET { VALUE = 2000; };
      ACTION = TerminateTask;
  };
};

COUNTER Counter1
{
  MAXALLOWEDVALUE = 0xffffffff; MINCYCLE = 0; TICKSPERBASE = 1;
  saPeriod = 1000;
};

ALARM Alarm1
{
```

```
      COUNTER = Counter1;
      ACTION = ACTIVATETASK { TASK = TaskA; };
      AUTOSTART = FALSE;
      saCyclic = TRUE { ALARMTIME = 0; CYCLETIME = 5; };
    };

    ALARM Alarm2
    {
      COUNTER = Counter1;
      ACTION = ACTIVATETASK { TASK = TaskB; };
      AUTOSTART = FALSE;
      saCyclic = TRUE { ALARMTIME = 0; CYCLETIME = 2; };
    };
};
```

DS-Design Tool shows that the application is clearly unscheduled, as the response generated by *TaskB* misses deadline:

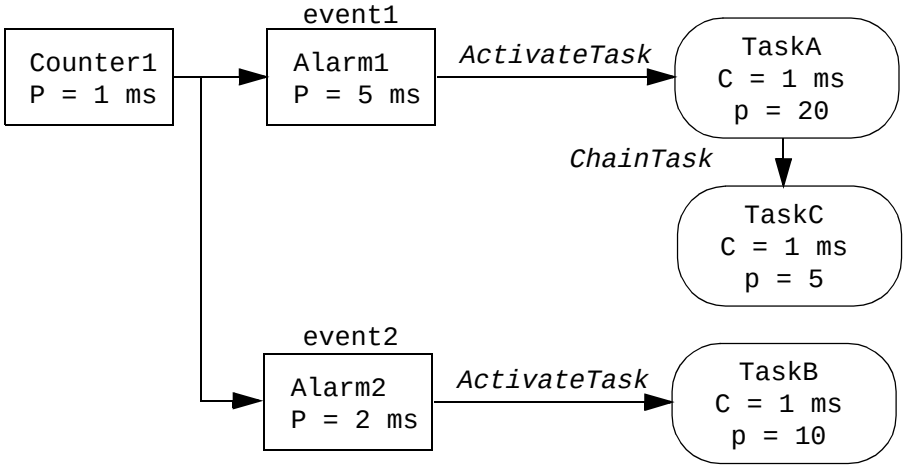
The details of response times at *Counter1/Alarm2/TaskB/TS_B* show the response time 3 ms, because the preemption time is 2 ms. Further details on preemption show that it is caused by 2 ms execution time of *TaskA*.

The details of response time at *Counter1/Alarm1/TaskA/TS_A* show “opposite” result. Response time is 2 ms - well below deadline, that is 4 ms. This response time is actually execution time of task sections *TS_1_1* and *TS_A* - each takes 1 ms. It is clear, that execution of task section *TS_1_1* can not be delayed, because in this case first deadline of *TaskA* will be missed.

However, execution of the task section *TS_A* is not so critical - the gap between the response time and deadline is 2 ms - quite enough time to execute *TaskB* and meet deadline at *Counter1/Alarm2/TaskB/TS_B*.

Clearly, *TaskA* uses processor time improperly for the second response, and it could decrease the priority for the computing the second response thus allowing *TaskB* to generate its response in time.

As there is no way to dynamic change of the task priority, this should be done statically. *TaskA* is split into two tasks: *TaskA*, that generates the first response, and *TaskC*, that generates the second response:



The key point here is the priority of *TaskC* - it is **lower** than the priority of *TaskB*.

The OIL file for the changed application is presented below. The DS-Design Tool specific parameters are in **bold**. The listing contains only lines that replace the lines in *italic* in previous listing:

```
TASK TaskA
{
  PRIORITY = 20;
  SCHEDULE = FULL;
  AUTOSTART = FALSE;
  ACTIVATION = 1;

  saTaskSection = SET { ID = "TS_A";
      WCET = 1000;
      DEADLINE = SET { VALUE = 1000; };
      ACTION = ChainTask { TASK = TaskC; };
  };
};

TASK TaskC
{
```

```
  PRIORITY = 5;
  SCHEDULE = FULL;
  AUTOSTART = FALSE;
  ACTIVATION = 1;

  saTaskSection = SET { ID = "TS_C";
      WCET = 1000;
      DEADLINE = SET { VALUE = 4000; };
      ACTION = TerminateTask;
  };
};
```

For the changed application the DS-Design Tool shows that all deadlines are met:

Indeed, the details of response times are as following:

- *Counter1/Alarm2/TaskB/TS_B*: response 2 ms, deadline 2 ms, deadline met
- *Counter1/Alarm1/TaskC/TS_C*: response 4 ms, deadline 4 ms, deadline met

The split of high-priority entity into chain of high-priority and low-priority entities is especially useful for Interrupt Service Routines. Priority of ISR in many cases can not be decreased, so the only option is to left to ISR only minimum execution, and move rest of it into the task, that is activated from ISR.

Restructuring Critical Sections

If deadline is missed because of blocking, it makes sense to explore where the blocking comes from. If it is due to critical section, the critical section may be made shorter, or may be split into two sections.

For example, in the application described in subsection Periodic Task, an ISR and the Resource, the high-priority task experiences the delay of execution due to the blocking caused by long critical section of low-priority task. If the blocking is too long, than the critical section of low-priority task could be redesigned or split.

Changing Scheduling Policy

Legacy real-time applications often employ non-preemptive scheduling as the implementation of non-preemptive scheduler usually takes less microprocessor resources. The drawback of the non-preemptive scheduler is the scheduling problems - i.e. the application misses deadlines.

In same cases the problem may be solved by means of using OSEKturbo mixed-preemptive scheduling.

For example, the application consisting of three periodic tasks is not scheduled when all tasks are non-preemptive (time units are milliseconds):

Task	Priority	Comp.Time	Period	Deadline
TaskA	10	1	3	3
TaskB	9	1	4	4
TaskC	8	2	7	7

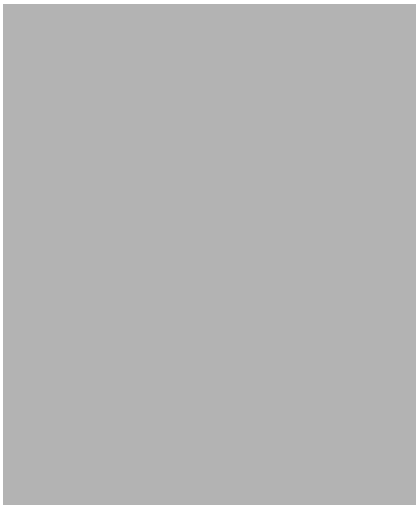
The OIL file for the application is presented below. The DS-Design Tool specific parameters are in **bold**. The objects and parameters that will be adjusted are in *italic*:

```
// Policy-Bad.oil

OIL_VERSION = "2.3";
#include "ost22mpc.oil"

CPU cpu1
{
```

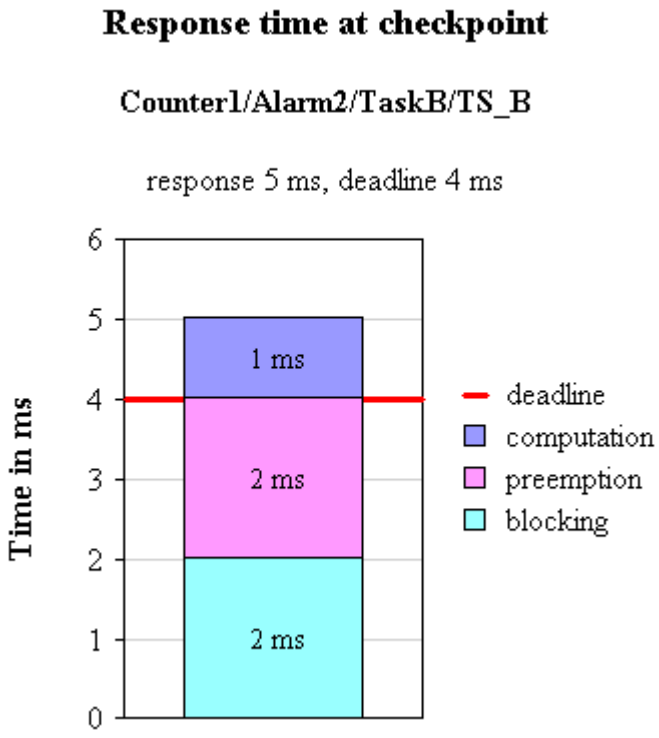
```
APPMODE app1;
OS os1
{
  TargetMCU = MPC555;
  STATUS = STANDARD;
  ResourceScheduler = FALSE;
  STARTUPHOOK = FALSE;SHUTDOWNHOOK = FALSE;
  PRETASKHOOK = FALSE;POSTTASKHOOK = FALSE;ERRORHOOK = FALSE;
  USEGETSERVICEID = FALSE; USEPARAMETERACCESS = FALSE;

  saTimeUnit = ms;
  saMeasureClockFrequency = 4000;
};

TASK TaskA
{
  PRIORITY = 10;
  SCHEDULE = NON;
  AUTOSTART = FALSE;
  ACTIVATION = 1;

  saTaskSection = SET { ID = "TS_A";
      WCET = 1;
      DEADLINE = SET { VALUE = 3; };
      ACTION = TerminateTask;
  };
};

TASK TaskB
{
  PRIORITY = 9;
  SCHEDULE = NON;
  AUTOSTART = FALSE;
  ACTIVATION = 1;

  saTaskSection = SET { ID = "TS_B";
      WCET = 1;
      DEADLINE = SET { VALUE = 4; };
      ACTION = TerminateTask;
  };
};

TASK TaskC
{
  PRIORITY = 8;
  SCHEDULE = NON;
  AUTOSTART = FALSE;
  ACTIVATION = 1;

  saTaskSection = SET { ID = "TS_C";
      WCET = 2;
      DEADLINE = SET { VALUE = 7; };
      ACTION = TerminateTask;
  };
```

```
    };

    COUNTER Counter1
    {
      MAXALLOWEDVALUE = 0xffffffff; MINCYCLE = 0; TICKSPERBASE = 1;
      saPeriod = 1;
    };

    ALARM Alarm1
    {
      COUNTER = Counter1;
      ACTION = ACTIVATETASK { TASK = TaskA; };
      AUTOSTART = FALSE;
      saCyclic = TRUE { ALARMTIME = 0; CYCLETIME = 3; };
    };

    ALARM Alarm2
    {
      COUNTER = Counter1;
      ACTION = ACTIVATETASK { TASK = TaskB; };
      AUTOSTART = FALSE;
      saCyclic = TRUE { ALARMTIME = 0; CYCLETIME = 4; };
    };

    ALARM Alarm3
    {
      COUNTER = Counter1;
      ACTION = ACTIVATETASK { TASK = TaskC; };
      AUTOSTART = FALSE;
      saCyclic = TRUE { ALARMTIME = 0; CYCLETIME = 7; };
    };
};
```

DS-Design Tool shows that application is unscheduled:



The reason is that the checkpoint *Counter1/Alarm2/TaskB/TS_B* misses deadline. DS-Design Tool shows the breakdown of the response time:



The preemption comes from *TaskA*. The blocking comes from *TaskC*:

Section	Time	Prio...	Type	Threshold
Counter1/Alarm3/TaskC/TS_C	2 ms	8	activate	10
Counter1/Alarm3/OS/SA_TERMINATE_TASK_DIS...	0	OS		
Total blocking	2 ms			

As the priority of *TaskA* can not be decreased, the only option is to decrease the blocking time by means of allowing preemption of *TaskC*.

The OIL file for the changed application is presented below. The DS-Design Tool specific parameters are in **bold**. The objects and parameters that are changed are in *italic*:

```
TASK TaskC
{
  PRIORITY = 8;
  SCHEDULE = FULL;
  AUTOSTART = FALSE;
  ACTIVATION = 1;

  saTaskSection = SET { ID = "TS_C";
      WCET = 2;
      DEADLINE = SET { VALUE = 7; };
      ACTION = TerminateTask;
  };
};
```

The blocking is completely gone, and application gets scheduled.

Employing OSEKturbo Internal Resources

OSEKturbo internal resources allow to improve scheduling of an application, and to get it scheduled if it is not. The explanation of this approach is beyond the scope of this manual[1]. The readers might find interesting to refer to article “Scheduling Fixed-Priority Tasks with Preemption Threshold”, by Yun Wang, Manas Saksena (see **[10]** in section

[1] To put it very simply, the intentional boost of task priorities allows to prevent preemption that helps when the low-priority task is close to the generation response and the preempting high-priority task would delay the response of low-priority task while deadline of high-priority task response is reasonably far.

"References,"). The example in this subsection was taken from the article, and only necessary changes were made.

In the example three periodic tasks have the following timing attributes (all times in milliseconds):

Task	Comp.Time	Period	Deadline
TaskA	20	70	50
TaskB	20	80	80
TaskC	35	200	100

Using Deadline Monotonic Algorithm, the task priorities are assigned as following:

Task	Priority
TaskA	4
TaskB	2
TaskC	1

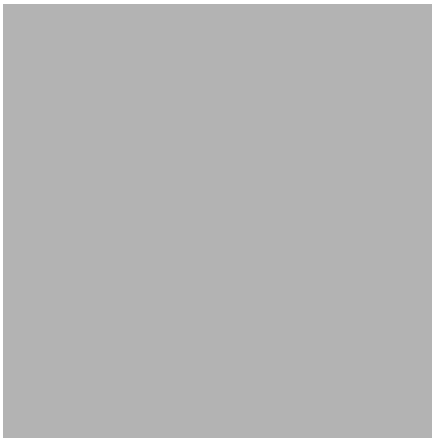
OIL file for the example is presented below. The DS-Design Tool specific parameters are in **bold**.

```
// InternalResources-Bad.oil

OIL_VERSION = "2.3";
#include "ost22mpc.oil"

CPU cpu1
{
    APPMODE app1;
    OS os1
    {
      TargetMCU = MPC555;
      STATUS = STANDARD;
      ResourceScheduler = FALSE;
      STARTUPHOOK = FALSE;SHUTDOWNHOOK = FALSE;
      PRETASKHOOK = FALSE;POSTTASKHOOK = FALSE;ERRORHOOK = FALSE;
      USEGETSERVICEID = FALSE; USEPARAMETERACCESS = FALSE;

      saTimeUnit = ms;
      saMeasureClockFrequency = 4000;
    };
```

```
TASK TaskA
{
  PRIORITY = 4;
  SCHEDULE = FULL;
  AUTOSTART = FALSE;
  ACTIVATION = 1;

  saTaskSection = SET { ID = "TS_A";
      WCET = 20;
      DEADLINE = SET { VALUE = 50; };
      ACTION = TerminateTask;
  };
};

TASK TaskB
{
  PRIORITY = 2;
  SCHEDULE = FULL;
  AUTOSTART = FALSE;
  ACTIVATION = 1;

  saTaskSection = SET { ID = "TS_B";
      WCET = 20;
      DEADLINE = SET { VALUE = 80; };
      ACTION = TerminateTask;
  };
};

TASK TaskC
{
  PRIORITY = 1;
  SCHEDULE = FULL;
  AUTOSTART = FALSE;
  ACTIVATION = 1;

  saTaskSection = SET { ID = "TS_C";
      WCET = 35;
      DEADLINE = SET { VALUE = 100; };
      ACTION = TerminateTask;
  };
};

COUNTER Counter1
{
  MAXALLOWEDVALUE = 0xffffffff; MINCYCLE = 0; TICKSPERBASE = 1;
  saPeriod = 1;
};

ALARM Alarm1
{
  COUNTER = Counter1;
  ACTION = ACTIVATETASK { TASK = TaskA; };
  AUTOSTART = FALSE;
  saCyclic = TRUE { ALARMTIME = 0; CYCLETIME = 70; };
```

```
    };

    ALARM Alarm2
    {
      COUNTER = Counter1;
      ACTION = ACTIVATETASK { TASK = TaskB; };
      AUTOSTART = FALSE;
      saCyclic = TRUE { ALARMTIME = 0; CYCLETIME = 80; };
    };

    ALARM Alarm3
    {
      COUNTER = Counter1;
      ACTION = ACTIVATETASK { TASK = TaskC; };
      AUTOSTART = FALSE;
      saCyclic = TRUE { ALARMTIME = 0; CYCLETIME = 200; };
    };
};
```

DS-Design Tool shows that application is unscheduled:



That proves the example description in the above mentioned article, that claims the application can not be scheduled for either preemptive or non-preemptive scheduling policy[1] (missed deadlines are in **bold**):

Task	Priority	Response for preemptive scheduling	Response for non-preemptive scheduling	Deadline
TaskA	4	20	**55**	50
TaskB	2	40	75	80
TaskC	1	**115**	75	100

The authors propose to boost run-time priorities for all tasks as following:

Task	Static Priority	Run-time Priority
TaskA	4	4
TaskB	2	4
TaskC	1	3

This may be achieved by means of using OSEKturbo internal resources. *TaskA* and *TaskB* share common internal resource *Res1*. To boost run-time priority of *TaskC*, the dummy task *TaskD* is introduced, that share common internal resource with *TaskC*. *TaskD* does nothing, and never gets scheduled.

OIL file for the modified example is presented below. The DS-Design Tool specific parameters are in **bold**. The added lines are in *italic*.

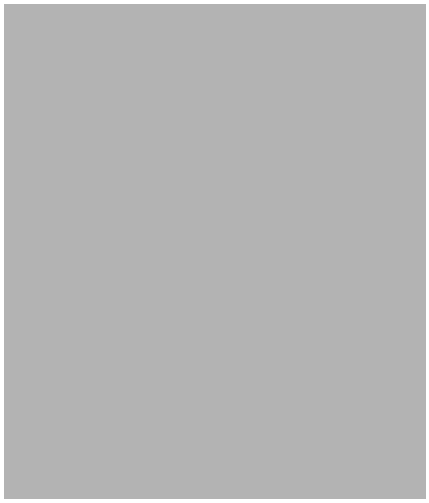
```
// InternalResources-Good.oil

OIL_VERSION = "2.3";
#include "ost22mpc.oil"

CPU cpu1
{
    APPMODE app1;
    OS os1
    {
      TargetMCU = MPC555;
      STATUS = STANDARD;
      ResourceScheduler = FALSE;
      STARTUPHOOK = FALSE;SHUTDOWNHOOK = FALSE;
      PRETASKHOOK = FALSE;POSTTASKHOOK = FALSE;ERRORHOOK = FALSE;
      USEGETSERVICEID = FALSE; USEPARAMETERACCESS = FALSE;

      saTimeUnit = ms;
      saMeasureClockFrequency = 4000;
    };

    RESOURCE Res1
    {
      RESOURCEPROPERTY = INTERNAL;
```

[1] For non-preemptive scheduling policy, the parameter SCHEDULE in all tasks should be set to NON.

```
};

TASK TaskA
{
  PRIORITY = 4;
  SCHEDULE = FULL;
  AUTOSTART = FALSE;
  ACTIVATION = 1;
  RESOURCE = Res1;

  saTaskSection = SET { ID = "TS_A";
      WCET = 20;
      DEADLINE = SET { VALUE = 50; };
      ACTION = TerminateTask;
  };
};

TASK TaskB
{
  PRIORITY = 2;
  SCHEDULE = FULL;
  AUTOSTART = FALSE;
  ACTIVATION = 1;
  RESOURCE = Res1;

  saTaskSection = SET { ID = "TS_B";
      WCET = 20;
      DEADLINE = SET { VALUE = 80; };
      ACTION = TerminateTask;
  };
};

RESOURCE Res2
{
  RESOURCEPROPERTY = INTERNAL;
};

TASK TaskC
{
  PRIORITY = 1;
  SCHEDULE = FULL;
  AUTOSTART = FALSE;
  ACTIVATION = 1;
  RESOURCE = Res2;

  saTaskSection = SET { ID = "TS_C";
      WCET = 35;
      DEADLINE = SET { VALUE = 100; };
      ACTION = TerminateTask;
  };
};

TASK TaskD
{
```

```
    PRIORITY = 3;
    SCHEDULE = FULL;
    AUTOSTART = FALSE;
    ACTIVATION = 1;
    RESOURCE = Res2;

    saTaskSection = SET { ID = "TS_D";
        WCET = 0;
        ACTION = TerminateTask;
    };
  };

  COUNTER Counter1
  {
    MAXALLOWEDVALUE = 0xffffffff; MINCYCLE = 0; TICKSPERBASE = 1;
    saPeriod = 1;
  };

  ALARM Alarm1
  {
    COUNTER = Counter1;
    ACTION = ACTIVATETASK { TASK = TaskA; };
    AUTOSTART = FALSE;
    saCyclic = TRUE { ALARMTIME = 0; CYCLETIME = 70; };
  };

  ALARM Alarm2
  {
    COUNTER = Counter1;
    ACTION = ACTIVATETASK { TASK = TaskB; };
    AUTOSTART = FALSE;
    saCyclic = TRUE { ALARMTIME = 0; CYCLETIME = 80; };
  };

  ALARM Alarm3
  {
    COUNTER = Counter1;
    ACTION = ACTIVATETASK { TASK = TaskC; };
    AUTOSTART = FALSE;
    saCyclic = TRUE { ALARMTIME = 0; CYCLETIME = 200; };
  };
};
```

Now the application is scheduled:



The response times are same as the article claims:

Task	Response for internal resources	Deadline
TaskA	40	50
TaskB	75	80
TaskC	95	100

Using Paths

The branches in tasks and ISRs code increase the number of the execution scenarios. Some of these scenarios are wrong. However, DS-Design Tool explores all scenarios, and therefore produces response time for wrong scenarios. Therefore, DS-Design Tool can tell that the deadline is missed while no real scenario leads to the checkpoint.

To exclude wrong scenarios, the paths mechanism of the DS-Design Tool should be used as it is described in the example in subsection Periodic Task, an ISR, and Paths.

The other advantage of paths mechanism is the possibility to set deadline for each path separately. In this case the *PATH* parameter should specify the path name in *DEADLINE* parameter.

Limitations for the Application Structure

The DS-Design Tool has the following particularities that imply the limitations for the applications:

- dynamic cyclic alarms (i.e. alarms that are set up in user's application) are considered as alarms statically started at the system startup;
- timing behavior of autostart of tasks is ignored;
- callback functions are not considered as standalone timing entity. Therefore, the timing behavior of the callback function should be described in a timing context of the task or and IST where the callback function is called;
- OSEKturbo system service *ActivateTask* can not be used in the critical sections framed by system services *GetResource* and *ReleaseResource*.

Dictionary of DS-Design Tool

For convenience of reading this subsection contains dictionary of the terms used within DS-Design Tool. Most of these terms are described in the subsections where details of the schedulability analysis of the applications are described.

Definition	Symbolic name	Description
Transaction	G	Set of application program entities, which are executed to respond to a specific kind of external events such as an interrupt request or counter advance independently from other transactions
Subtransaction	g	Part of transaction that combines a subset of tasks, which may be connected by precedence relations
Task	T	Implementation of sequential algorithm managed by operating system as a separate execution entity during scheduling process. It is OSEKturbo task.
Basic task	B	Task that can never go into waiting state
Extended task	E	Task that may have waiting state
Task section	t	Part of task delimited by operators affecting on scheduling process
ISR	I	Interrupt service routine. It is OSEKturbo ISR

ISR section	i	Part of interrupt service routine delimited by operators affecting on scheduling process
Subtask	e	Part of extended task delimited by wait event operators
Timepoint	x	End of task section or ISR section (where task or ISR generate response)
Checkpoint	c	Timepoint that has deadline set
Execution graph	X	Directed graph of the tasks sections and the ISR sections in a subtransaction. Execution graph represents all routes in the subtransaction
Execution scenario	S	Route, or path on the execution graph, that the subtransaction could execute
Worst-case execution time	C	Worst-case execution time of an entity, needed to complete computation assuming that processor is fully allocated to this entity
Deadline	D	Deadline for the task section or ISR section relative to the start of subtransaction that comprises this task section or ISR section
Execution path	W	Path of execution of task sections and ISR sections that form an execution scenario
Minimal interarrival time	M	Minimal time of arrival (release) of an entity (usually it is time between consequential occurrences of an interrupt service routine)
Jitter	J	The time variation of the release of an entity (usually it is time variation of the ISR release)
Period	P	Period of releases of periodic entities
Priority	p	Priority of the task or the ISR

7

Analysis of ECC Applications

This chapter consists of the following sections:

- General
- Particularities of Waiting State
- Applying Computational Model
- Tuning an Application
- Limitations for Application Structure

General

The applications of Extended Conformance Classes usually employ OSEKturbo extended tasks. Extended tasks may have **waiting state**, which in most OSEKturbo applications is used to wait for several application OSEKturbo events.

Particularities of Waiting State

Call of OSEKturbo operating system *WaitEvent* could imply waiting state, and is a rescheduling point therefore. Hence, the task code that ends with a call to *WaitEvent* system service needs to be a task section. In a similar way the task or the ISR code that ends with a call to *SetEvent* system service, should be a section as well.

DS-Design Tool checks that task that could wait for OSEKturbo events is activated in the application - otherwise DS-Design Tool believes that the task is in suspended state, and can not wait for events.

Quite often the extended task waits for the several OSEKturbo events that are set in different tasks and interrupt service routines. Therefore, the extended task becomes the member of several subtransactions.

Meeting deadlines in applications of Extended Conformance class is more difficult as one task could serve events that have different deadlines. Therefore, in many cases the task delays the response to the event with a short deadline because it computes response to the event that has longer deadline. From scheduling analysis point of view, the task preempts itself when it is executed as part of different subtransactions.

The other scheduling problem of extended tasks is *multi-rate* priority assignment. Priority allocation algorithms such as RMA and DMA assign task priority depending on deadline that the task has. As the extended task may generate response to several event that have various deadlines, the priority allocation becomes non-trivial.

In many cases the extended task builds an execution loop around *WaitEvent* operating system call. The loop starts with the wait on events, checks the events state, and handles each event. Then the task repeats the loop. DS-Design Tool employs specific action of the task section that is called *CheckEvent* to build the conditional handling of events. Next sections demonstrate the use of this action. *CheckEvent* action is similar to *GoTo* action described in ***Analysis of BCC Applications***, because the action does not relate to the operating system call, but is aimed to provide DS-Design Tool with the information about execution scenarios. *CheckEvent* action is kind of conditional goto operator. The action points to two task sections - first section is executed when the checked event is set, and the second section is executed when the checked event is cleared.

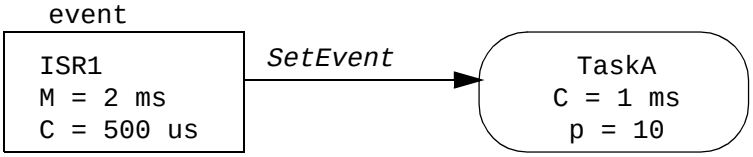
Applying Computational Model

In general, the steps described in **Applying Computational Model** for BCC applications should be applied to the applications with the extended tasks. This is demonstrated on the examples below.

Extended Task with Single Event and an ISR

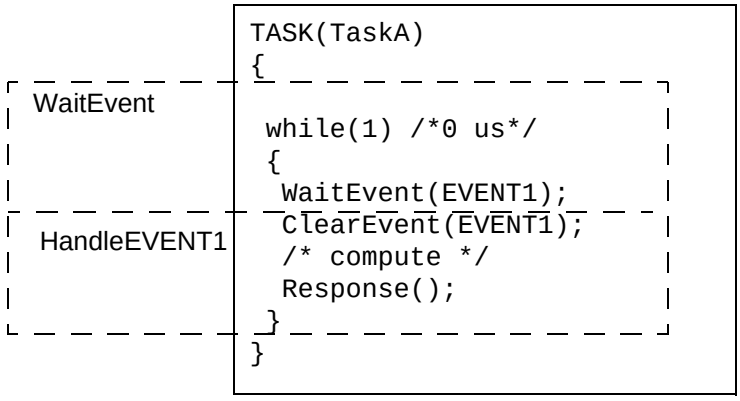
The application consists of one extended task that waits on single OSEKturbo event. The event is set in the interrupt service routine.

The scheme of the application is as following:

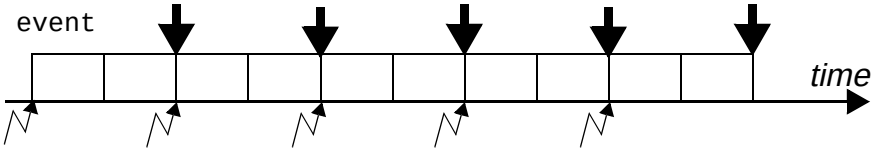


Task *TaskA* is activated during system startup. The task goes into endless loop waiting on an event *EVENT1*. When *EVENT1* is set, the *TaskA* computes the response within 1 ms. The deadline for the response is 2 ms. The *EVENT1* is set by an interrupt service routine *ISR1* that has minimal interarrival time 2 ms. Worst-case execution time of the *ISR1* is 500 us.

The scheme of task code is as following:



Step 1. Timing requirements of applications events are presented on the picture:



Event arrives every 2 ms, and has deadlines 2 ms.

Step2. Application has one transaction:

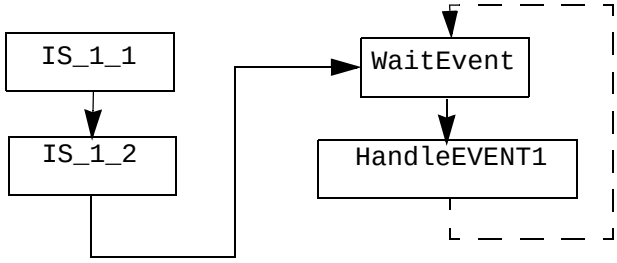
- *ISR1* that has minimal interarrival time 2 ms

Step3. Application subtransactions:

- Transaction *ISR1* has single subtransaction *ISR1* (full name *ISR1/ISR1*)

Step 4. The execution graphs:

- subtransaction *ISR1/ISR1*:



The subtransaction consists of the *ISR1* and the *TaskA* (more precisely, two ISR sections of ISR1, identified as *IS_1_1* and *IS_1_2*, and single task section *HandleEVENT1* of *TaskA*). ISR section *IS_1_1* is an initial section of *ISR1*. Note that task section *WaitEvent* is not counted in the subtransaction, because it is considered as executed beyond response generation.

If deadline is set for the section *WaitEvent*, then the end of sections will be considered as a checkpoint, and the section worst-case execution time will be counted in the response time.

Step 5. The following execution scenarios exists:

Execution scenario for subtransaction *ISR1/ISR1*:

- *IS_1_1*, *IS_1_2*, *HandleEVENT1*

Step 6. Tasks and ISR sections are as following:

- *ISR1/IS_1_1*, that ends with an action *SetEvent* of event *EVENT1* for task *TaskA*, and is followed by the section *IS_1_2*. Worst-case execution time of the section is 300 us
- *ISR1/IS_1_2*, that ends with an action *LeaveISR*. Worst-case execution time of the section is 200 us
- *TaskA/WaitEvent*, that ends with two *WaitEvent* action. Worst-case execution time of the section is considered as zero

- *TaskA/HandleEVENT1*, that ends with *GoTo* action to *WaitEvent* section. Worst-case execution time of the section is 1000 us

Step 7. The checkpoints are as following:

- Checkpoint is at the end of *HandleEVENT1* action of *TaskA*, when the task generates first response of the subtransaction. The deadline is 2 ms

Step 8.OIL file for the example is presented below. The DS-Design Tool specific parameters are in **bold**.

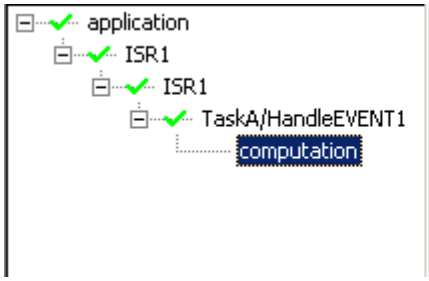
```
// ExtendedTaskAndISR.oil

OIL_VERSION = "2.3";
#include "ost22mpc.oil"

CPU cpu1
{
    APPMODE app1;
    OS os1
    {
      TargetMCU = MPC555;
      STATUS = STANDARD;
      ResourceScheduler = FALSE;
      STARTUPHOOK = FALSE;SHUTDOWNHOOK = FALSE;
      PRETASKHOOK = FALSE;POSTTASKHOOK = FALSE;ERRORHOOK = FALSE;
      USEGETSERVICEID = FALSE; USEPARAMETERACCESS = FALSE;
      IsrStackSize = 0x200;

      saTimeUnit = us;
      saMeasureClockFrequency = 4000;
    };

    EVENT EVENT1 { MASK = 1; };

    TASK TaskA
    {
      PRIORITY = 10;
      SCHEDULE = FULL;
      AUTOSTART = TRUE;ACTIVATION = 1;
      STACKSIZE = 0x100;
      EVENT = EVENT1;

      saInitial = "WaitEvent";
      saTaskSection = SET { ID = "WaitEvent";
          WCET = 0;
          ACTION = WaitEvent { EVENT = EVENT1;
                               NEXT = "HandleEVENT1"; };
      };
      saTaskSection = SET { ID = "HandleEVENT1";
          WCET = 1000;
```

```
            ACTION = GoTo { NEXT = "WaitEvent"; };
            DEADLINE = SET { VALUE = 2000; };
        };
    };

    ISR ISR1
    {
      CATEGORY = 2;
      PRIORITY = 10;
      saMinIATime = 2000;

      saInitial="IS_1_1";
      saISRSection = SET { ID = "IS_1_1";
          WCET = 300;
          ACTION = SetEvent { NEXT = "IS_1_2";
                              TASK = TaskA; EVENT = EVENT1; };
      };
      saISRSection = SET { ID = "IS_1_2";
          WCET = 200;
          ACTION = LeaveISR;
      };
    };
};
```

Step 9. Run DS-Design Tool.

Step 10. The result of the analysis are explained below.

The transactions and checkpoints of the application are as following:



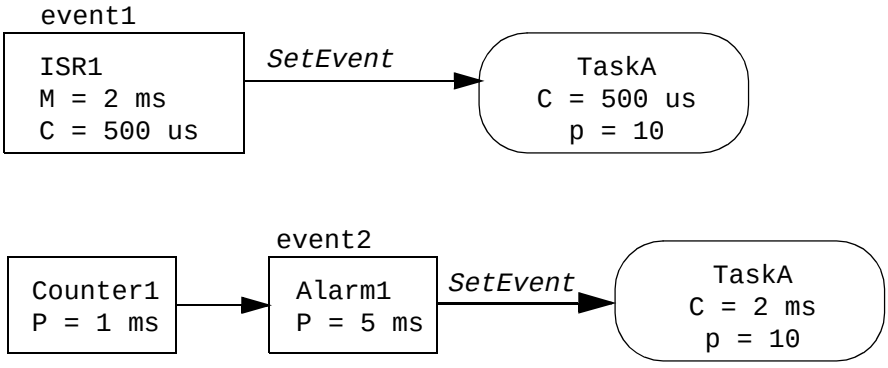
Section	Time	Priority	Type
CURRENT JOB			
SA_ENTER_ISR	0	OS	
ISR1/IS_1_1	300 µs	10	activate
SA_ISR_SET_EVENT	0	OS	
ISR1/IS_1_2	200 µs	10	return
SA_LEAVE_ISR	0	OS	
TaskA/HandleEVENT1	1 ms	10	activate
Total computation	1.5 ms		

Application meets deadline as response time is 1.5 ms while deadline is 2 ms.

Extended Task with Two Events

The application consists of one extended task that waits for two OSEKturbo events. One event is set in the interrupt service routine, another is a periodic event set by an alarm.

The scheme of the application is as following:



Task *TaskA* is activated during system startup. The task goes into endless loop waiting on the events *EVENT1* and *EVENT2*.

When *EVENT1* is set, the *TaskA* computes the response within 500 us. The deadline for the response is 2 ms. The *EVENT1* is set by an interrupt service routine *ISR1* that has minimal interarrival time 2 ms. Worst-case execution time of the *ISR1* is 500 us.

When *EVENT2* is set, the *TaskA* computes the response within 2000 us. The deadline for the response is 5 ms. *EVENT2* is set by alarm *ALARM1* that has minimal interarrival time 5 ms.

NOTE The scheme and timing requirements of the application are similar to ones presented in example **Periodic Task and an ISR**. However, as will be shown, when one extended task serves two events, application is unscheduled.

The scheme of task code is as following:

```
TASK(TaskA)
{
EventMaskType evt;
 while(1)
 {
 WaitEvent(EVENT1|EVENT2);
  GetEvent(&ev);
  if(ev & EVENT1)
  { /* 500 us */
    ClearEvent(EVENT1);
    /* compute */
    Response1();
    continue;
  }
  if(ev & EVENT2)
  { /* 2 ms */
    ClearEvent(EVENT2);
    /* compute */
    Response2();
    continue;
  }
 } /* end-while */
}
```

WaitEvent

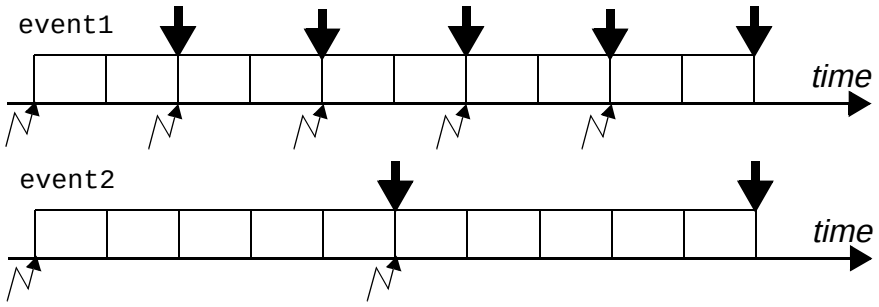
CheckEVENT1

HandleEVENT1

CheckEVENT2

HandleEVENT2

Step 1. Timing requirements of applications events are presented on the picture:



Event1 arrives every 2 ms, and has deadlines 2 ms.

Event2 arrives every 5 ms, and has deadlines 5 ms.

Step2. Application has two transactions:

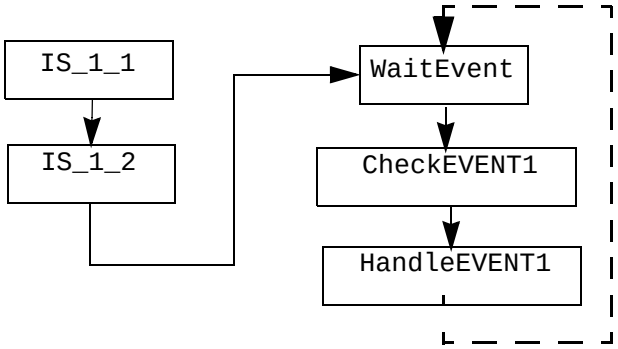
- *ISR1* that has minimal interarrival time 2 ms
- *Counter1* that has period 1 ms

Step3. Application subtransactions:

- Transaction *ISR1* has single subtransaction *ISR1* (full name *ISR1/ISR1*)
- Transaction *Counter1* has single subtransaction *Alarm1* (full name *Counter1/Alarm1*)

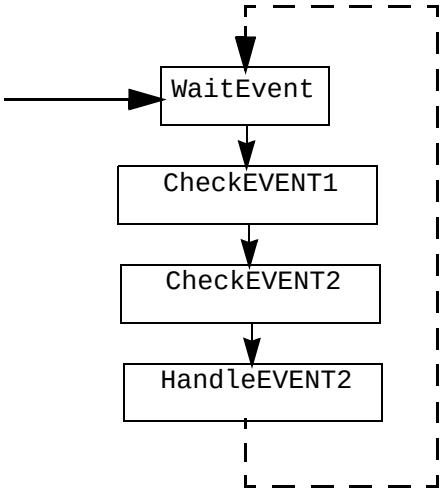
Step 4. The execution graphs:

- subtransaction *ISR1/ISR1*:



The subtransaction consists of *ISR1* and *TaskA* (more precisely, two ISR sections of ISR1, identified as *IS_1_1* and *IS_1_2*, and two task sections of *TaskA*, identified as *CheckEVENT1* and *HandleEVENT1*). ISR section *IS_1_1* is an initial section of *ISR1*.

- subtransaction *Counter1/Alarm1*:



The subtransaction consists of *TaskA* (more precisely, three task sections of *TaskA*, identified as *CheckEVENT1*, *CheckEVENT2*, and *HandleEVENT2*).

Note that task section *WaitEvent* is not counted in the subtransactions, because it is considered as executed beyond the response generation.

The task *TaskA* belongs to two subtransactions of two different transactions. Moreover, the task section *CheckEvent1* also belongs to two subtransactions of two different transactions.

Step 5. The following execution scenarios exists:

Execution scenario for subtransaction *ISR1/ISR1*:

- *IS_1_1, IS_1_2, CheckEVENT1, HandleEVENT1*

Execution scenario for subtransaction *Counter1/Alarm1*:

- *CheckEVENT1, CheckEVENT2, HandleEVENT1*

Step 6. Tasks and ISR sections are as following:

- *ISR1/IS_1_1*, that ends with action *SetEvent* of event *EVENT1* for task *TaskA*, and is followed by section *IS_1_2*. The worst-case execution time of the section is 300 us
- *ISR1/IS_1_2*, that ends with action *LeaveISR*. The worst-case execution time of the section is 200 us
- *TaskA/WaitEvent*, that ends with two *WaitEvent* action. The worst-case execution time of the section is considered as zero
- *TaskA/CheckEVENT1*, that ends with *CheckEvent* action to jump to *HandleEVENT1* section if event *EVENT1* is set. If this event is cleared, then the action jumps to *CheckEVENT2* section. The worst-case execution time of the section is considered as zero
- *TaskA/HandleEVENT1*, that ends with *GoTo* action to *WaitEvent* section. Worst-case execution time of the section is 500 us
- *TaskA/CheckEVENT2*, that ends with *CheckEvent* action to jump to *HandleEVENT2* section if event *EVENT2* is set. If this event is cleared, then the action jumps to *WaitEvent* section. The worst-case execution time of the section is considered as zero
- *TaskA/HandleEVENT2*, that ends with *GoTo* action to *WaitEvent* section. The worst-case execution time of the section is 2 ms

Step 7. The checkpoints are as following:

- Checkpoint is at the end of *HandleEVENT1* action of *TaskA*, when the task generates first response of subtransaction *ISR1/ISR1*. The deadline is 2 ms

- Checkpoint is at the end of *HandleEVENT2* action of *TaskA*, when the task generates first response of the subtransaction *Counter1/Alarm1*. The deadline is 5 ms

Step 8.OIL file for the example is presented below. The DS-Design Tool specific parameters are in **bold**.

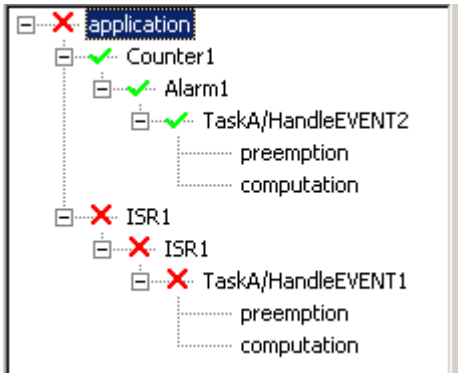
```
// ExtendedTaskISRAndAlarm.oil

OIL_VERSION = "2.3";
#include "ost22mpc.oil"

CPU cpu1
{
    APPMODE app1;
    OS os1
    {
      TargetMCU = MPC555;
      STATUS = STANDARD;
      ResourceScheduler = FALSE;
      STARTUPHOOK = FALSE;SHUTDOWNHOOK = FALSE;
      PRETASKHOOK = FALSE;POSTTASKHOOK = FALSE;ERRORHOOK = FALSE;
      USEGETSERVICEID = FALSE; USEPARAMETERACCESS = FALSE;
      IsrStackSize = 0x200;

      saTimeUnit = us;
      saMeasureClockFrequency = 4000;
    };

    EVENT EVENT1 { MASK = 1; };
    EVENT EVENT2 { MASK = 2; };

    TASK TaskA
    {
      PRIORITY = 10;
      SCHEDULE = FULL;
      AUTOSTART = TRUE;ACTIVATION = 1;
      STACKSIZE = 0x100;
      EVENT = EVENT1; EVENT = EVENT2;


      saInitial = "WaitEvent";
      saTaskSection = SET { ID = "WaitEvent";
          WCET = 0;
          ACTION = WaitEvent { EVENT = EVENT1; EVENT = EVENT2;
                               NEXT = "CheckEVENT1"; };
      };
      saTaskSection = SET { ID = "CheckEVENT1";
          WCET = 0;
```

```
            ACTION = CheckEvent { EVENT = EVENT1;
                                  CLEARED = "CheckEVENT2";
                                   SET = "HandleEVENT1"; };
          };
          saTaskSection = SET { ID = "HandleEVENT1";
              WCET = 500;
              DEADLINE = SET { VALUE = 2000; };
              ACTION = GoTo { NEXT = "WaitEvent"; };
          };
          saTaskSection = SET { ID = "CheckEVENT2";
              WCET = 0;
              ACTION = CheckEvent { EVENT = EVENT2;
                                    CLEARED = "WaitEvent";
                                     SET = "HandleEVENT2"; };
          };
          saTaskSection = SET { ID = "HandleEVENT2";
               WCET = 2000;
               DEADLINE = SET { VALUE = 5000; };
               ACTION = GoTo { NEXT = "WaitEvent"; };
          };
        };

        ISR ISR1
        {
          CATEGORY = 2;
          PRIORITY = 10;
          saMinIATime = 2000;

          saInitial="IS_1_1";
          saISRSection = SET { ID = "IS_1_1";
              WCET = 300;
              ACTION = SetEvent { NEXT = "IS_1_2";
                                  TASK = TaskA; EVENT = EVENT1; };
          };
          saISRSection = SET { ID = "IS_1_2";
              WCET = 200;
              ACTION = LeaveISR;
          };
        };

        COUNTER Counter1
        {
          MAXALLOWEDVALUE = 0xffffffff; MINCYCLE = 0; TICKSPERBASE = 1;
          saPeriod = 1000;
        };
        ALARM Alarm1
        {
          COUNTER = Counter1;
          ACTION = SETEVENT { TASK = TaskA; EVENT = EVENT2; };
          AUTOSTART = TRUE { ALARMTIME = 0; CYCLETIME = 5; };
        };
    };
```

Step 9. Run DS-Design Tool.

Step 10. The result of the analysis is explained below.

The transactions and checkpoints of the application are as following:



Note that for *ISR1* event is not met - the response time is 3.5 ms while deadline is 2 ms. This is due to preemption from the subtransaction *Counter1/Alarm1* (2 ms) and the next arrival of *ISR1* (500 us).

The example shows that the single extended task that serves the events of different deadlines experiences scheduling difficulties because it has no mean to prioritize responses.[1]

Tuning an Application

Most methods considered for the BCC applications are applicable.

When exploring analysis results, the great attention should be paid to the preemption structure. If deadline is missed due to preemption caused by the same task as the task that is being preempted (as demonstrated in the example **Extended Task with Two Events**), then it makes sense to introduce new basic task that serves the event with a short deadline.

1. In some respect this extended task *TaskA* acts as two non-preemptive tasks of same priority.

Limitations for Application Structure

DS-Design Tool can not perform robust analysis of extended tasks that are dynamically activated and terminated. Therefore, the application is assumed to start all extended tasks, and they should not be terminated.

8

Advanced Features

This chapter consists of the following sections:

- General
- Internal Resources
- Using text views
- Fine Tuning of Applications
- Batch Mode of Analysis
- Dealing with Operating System overhead

General

This section contains description of advanced features of DS-Design Tool that might be useful for more comprehensive analysis and for better satisfying of organizational software development process.

Internal Resources

Internal resources in OSEKturbo are primarily aimed to provide groups of mutually non-preemptable tasks and co-operative tasks (see **[9]** in section "References,"). However, as mentioned in **Section Employing OSEKturbo Internal Resources**, they impact scheduling of applications as scheduling behaves very similar to so-called *scheduling with preemption threshold* that is described in number of sources. (For example, in the paper Scheduling Fixed-Priority Tasks with Preemption Threshold, by Yun Wang and Manas Saksena - see **[10]** in section "References,").

Scheduling with internal resources may be considered as fully preemptive when each task has exactly one internal resource which ceiling priority is the same as task priority, or non-preemptive - when all tasks have internal resource that is RES_SCHEDULER indeed. As both preemptive and non-preemptive schedulings have advantages and disadvantages, scheduling

with internal resources may provide benefits of both kind of scheduling policies. Therefore, scheduling with internal resources may be used to improve application scheduling.

Using text views

Graphic views of GraViTool are handy to explore and compare for most applications. However, when an application is mature, and schedulability analysis becomes a routine, the application designer might find more useful to look at *text views* generated by the GraViTool.

Text views contain almost the same amount of information as charts, but they are plain text files. Therefore, an experienced developer may apply text processing utilities to quick finding of precise information.

TIP Text views are extremely useful for making reports on application scheduling. Just insert all file, or cut and paste fragments of it into reporting document.

Text views are generated by GraViTool by means of using menu item "File|Generate Text Views...", or by means of clicking diskette-icon on the toolbar. In the other case the save file dialog window appears prompting to specify the name of the text views file. By default it is name of the file currently opened in GraViTool with extension ".txt" instead of file extension ".saml".

Here is an example of text views for the application described in **Single Periodic Task**[1]:

```
 1   Application: application
 2   Verdict: scheduled
 3
 4   List of transactions
 5   Transaction, minimal inter-arrival time:
 6     Counter1,  1 ms
 7
 8     Transaction Counter1, list of subtransactions
 9     Subtransaction, Every, Offset, Jitter:
10       Counter1/Alarm1, 5, 0, 0
```

1. Line numbers are not included into text views file. The lines are numbered by "pr" utility.

```
11
12       Subtransaction Alarm1, list of checkpoints
13       Checkpoint, verdict, response time, deadline, arrival:
14         Counter1/Alarm1/TaskA/TS_A, meet,  2 ms,  5 ms, 0
15
16         computation at checkpoint Counter1/Alarm1/TaskA/TS_A
17         Section, Time, Priority, Type:
18           TOTAL,  2 ms
19           current_job,  2 ms
20           Counter1/Alarm1/TaskA/TS_A,  2 ms, 10, activate
21
22
23
24
25     Total CPU utilization:  40 %
26       Counter1/Alarm1,  40 %
27
28       utilization per subtransaction Alarm1
29         TaskA,  40 %
```

The first line of the file contains the name of application. Second line specifies if applcation is scheduled. Then the lists of transactions and subtransactions follow.

For checkpoint it is specified if deadline is met ("*meet*" in line 14). If deadline is not met, it is tagged with "*miss*" tag.

TIP Text-processing utilities from Unix world such as *grep*, *awk*, *wc*, etc. can be used to extract desired information from text views file. For example, to count number of missed deadlines in application, one could use *grep miss text_views_file | wc -l*

Fine Tuning of Applications

While DS-Design Tool is primarly intended to work in as integrated part of OSEK Builder, there are advantages of using components of DS-Design Tool separately.

For example, if it is a need to change slightly the application timing information, and compare scheduling results of original application with changed one, it makes sense to store scheduling analysis data in different files, and then launch two instances of graphical views tool to explore both files and compare them.

The alternative way of comparing results of analysis is to produce text views, and compare two or more text files by means of using editors and utilities that visualize differnces in text files.

Batch Mode of Analysis

As DS-Design Tool uses OSEKturbo System Generator, analysis may be run in batch mode. For example, the following command line of Command Prompt directs system generator to perform schedulability analysis of application SinglePeriodicTask.oil:

```
sysgen -b -z SinglePeriodicTask.oil.saml SinglePeriodicTask.oil
```

The system generators reads and verifies an OIL-file of the application *`SinglePeriodicTask.oil`*, and produces graphical views file *`SinglePeriodicTask.oil.saml`*, if no errors were found in the OIL file. In this case the return code of System Generator is zero. If OIL file contains errors, then no graphical views file is produced, and error code is not zero.

If there is no file name specified after -z option, the default file name will be used, which is combined from an input OIL file name, and has added extension *.saml*.

Note that even application is not scheduled, return code of System Generator is zero, as OIL file is valid.

The graphical views tool GraViTool can be started in command line as well. For example, to view the results of the analysis for the application mentioned above, the following command line of Command Prompt may be used:

```
GraViTool SinglePeriodicTask.oil.saml
```

Dealing with Operating System overhead

Operating System overhead is an important impact on application scheduling. In the embedded applications the execution times of tasks and ISRs are sometimes comparable with OS system services timing.

The DS-Design Tool uses set of OIL parameters to identify timing of OSEKturbo services. These parameters are summarized in OIL Language Quick Reference. While OS overhead parameters are not intended for use of the application designer, in subtle cases the values of these parameters may be adjusted for better characterizing of particularities of OSEKturbo operating system configuration and hardware timing. Example of overhead numbers are provided as part of SA sample included into OSEKturbo OS installation. OS timings depend on OS configuration, therefore they should be measured for the user configuration to get accurate results.

9

Troubleshooting

This chapter consists of the following sections:

- Evaluation of Timing Parameters
- Eliminating Wrong Application Structure
- Looking at Intermediate Data Files
- Computation Issues
- Full Utilization of Processor
- Blocking Pessimism
- Known Problems

In this section some advice is given which may be useful for developers working with DS-Design Tool.

Evaluation of Timing Parameters

More accurate are the timing parameters values, more realistic are the results produced by DS-Design Tool. Therefore, significant efforts should be put into evaluating and tracking changes of timing parameters. For example, worst-case execution time can be measured by means of employing DS-V tool, or by means of using logic analyzer or time measurement hardware, or by means of counting instructions timing. All methods vary in efforts and accuracy.

Note that exactly the same application may exhibit different timing behavior on various hardware configuration (e.g. number of memory wait cycles impacts timing enormously).

In complex cases several methods can be used, and then the most appropriate (accurate) should be chosen.

As application code changes, the timing parameters need to be re-assessed. This can be done on regular basis, or on specified steps of development process. For example, at integration phase all important timing parameters can be re-assessed and measured.

Eliminating Wrong Application Structure

DS-Design Tool may discover problems in application structure. There is no standard means in OSEK/VDX Specifications to have a guarantee consistency and completeness of the applications. For example, in standard error status most run-time errors get unnoticed.

As DS-Design Tool forces the application developers to look carefully at the semantics of the application, the unexpected semantics errors could be discovered, which do not relate directly to DS-Design Tool specifics.

For example, DS-Design Tool could discover improper use of system services in short critical sections that are framed by pairs of OSEKturbo Fast Disable/Enable API functions.

Looking at Intermediate Data Files

DS-Design Tool uses intermediate files that contain a lot of useful information. For example, graphical views file which is produced by System Generator is the XML file, and it contains all information that GraViTool visualizes. In case of troubles with visualization it could make sense to look at graphical views file (e.g. by means of using Internet Explorer), and try to understand the source of problem.

TIP Graphical Views files or their fragments could be useful for support team to find out the issues with the tool or an application.

There is also “undocumented” feature of System Generator. When launched with option “-x filename”, it produces detailed report of analysis steps in the file filename. This file can be also delivered to support team in case problems arrive.

Computation Issues

Careful selection on measurement units helps to avoid, or at least minimize computations rounding. For example, if the task execution time is 1234 microseconds, and time unit in OIL file is set to milliseconds, then the DS-Design Tool will believe, the execution time is 1 ms, or 1000 microseconds, introducing almost 25% of error.

When several of such values are used in computation, then the rounding error could grow dramatically.

Hence, the time units should adequately represent accuracy of measurement. It is recommended to use time unit of less granularity and bigger number therefore in order to minimize rounding impact.

The other issue is the relations between the execution times of different entities. For example, if tasks have execution times measured in seconds, while Operating System services take few microseconds, then the Operating System overhead could be small enough to be visualized while the impact on scheduling can not be ignored. In such cases GraViTool could show the missed deadlines while the reason of this is unnoticed. In such cases methods described in Looking at Intermediate Data Files may be worth trying.

Full Utilization of Processor

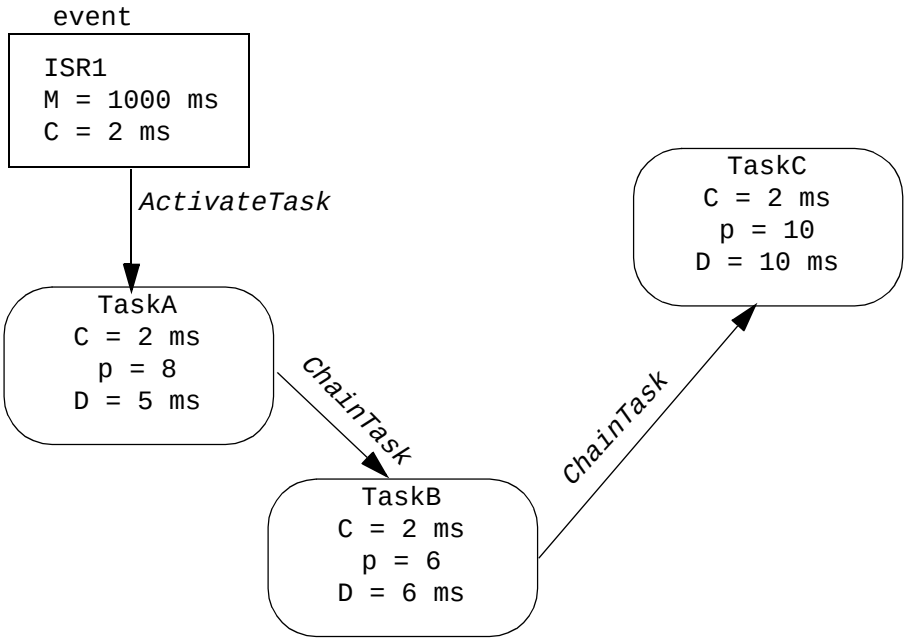
If DS-Design Tool encounters that processor utilization reaches 100%, it does not compute checkpoints that are on this boundary and above. In other words, DS-Design Tool takes pessimistic approach for fully utilized processor. This is why the application that are schedulable at 100% utilization are considered as non-schedulable by DS-Design Tool. The reason for that is the fact that any small deviation in timing behavior leads to unschedulable application while theoretically the application may be considered as being scheduled.

Blocking Pessimism

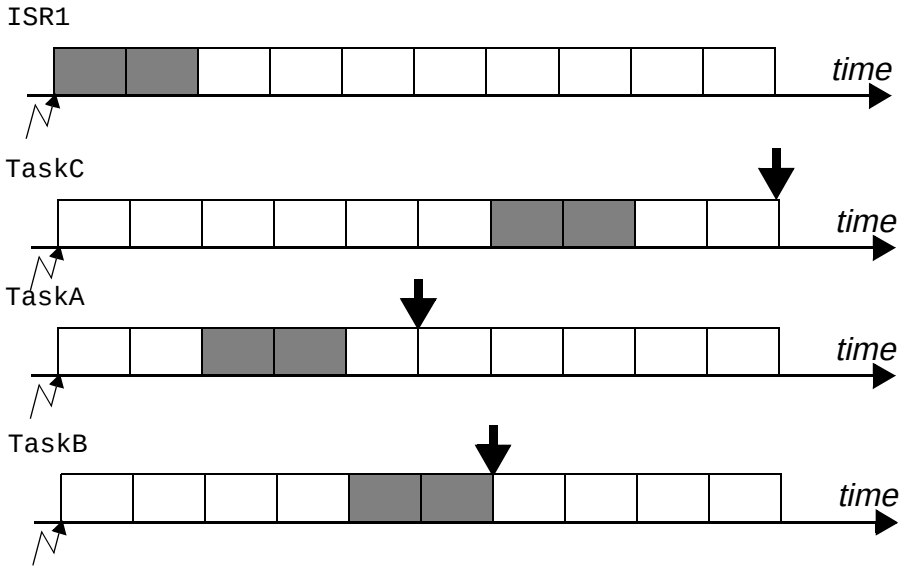
Another are where DS-Design Tool takes pessimistic approach is blocking possibly caused by the previous jobs of the same subtransaction.

For example, the application that consists of single transaction and single subtransaction, may exhibit blocking even if it looks like it should not be there.

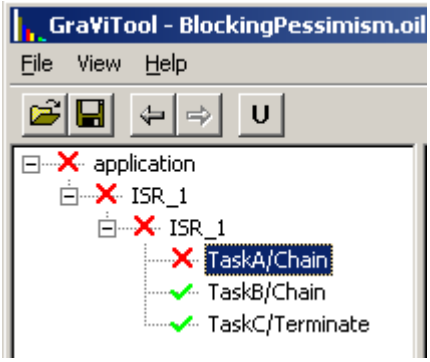
The application on the picture below consists of one ISR *ISR1* and three tasks that build a chain:



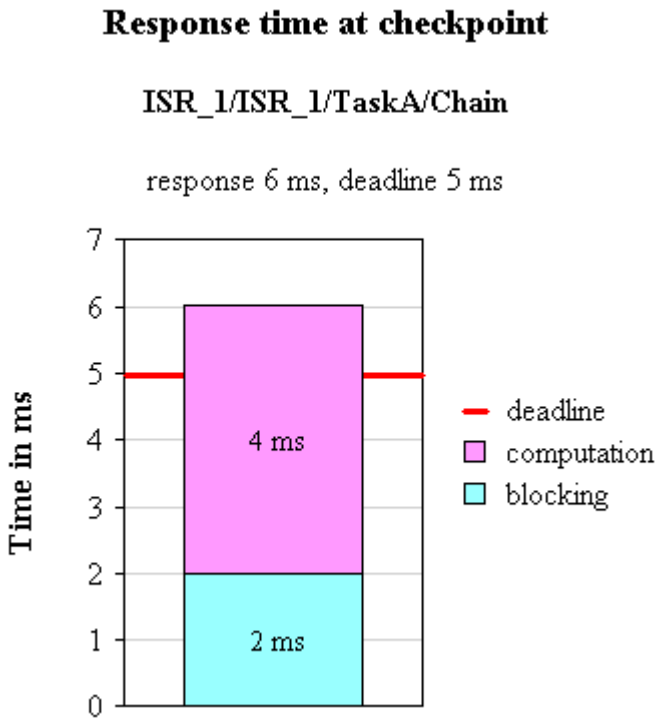
TaskA has middle priority, *TaksB* has low priority, and *TaskC* has high priority. At first glance, the timing diagram looks like the following one (ordered by priorities):



All tasks meet deadlines, and application is scheduled. However, DS-Design Tool provides different result:



The application is not scheduled because checkpoint at *TaskA* misses deadline. The response time at this checkpoint is as following:



The checkpoint experiences 2 ms blocking that comes from high priority *TaskC*:

Section	Time	Priority	Type	Threshold
ISR_1/ISR_1/TaskC/Terminate	2 ms	10	activate	10
Total blocking	2 ms			

Though *TaskC* runs after *TaskA* fully completes execution, DS-Design Tool assumes, that next arrival of *ISR1* overlaps start of the execution of *TaskC* and the blocking from *TaskC* occurs therefore.

For simplified case like the presented above this is virtually unrealistic scenario and it should not be considered. Nevertheless, DS-Design Tool considers it by means of applying `blocking pessimism` approach, because in complex cases this approach provides a mean to deal with blocking.

Known Problems

The problems are unknown before the time of the release of this document. Please refer to Technical Support Information for updates.

A

Sample Application

This chapter consists of the following sections:

- Description
- Source Files
- Results of Analysis

Description

The standard OSEKturbo sample is used to demonstrate how to apply DS-Design Tool to analyze timing behavior of an application.

The standard sample is described in OSEKturbo Operating System Technical Reference. In this section only DS-Design Tool related changes are described.

TIP Reading sample description in OSEKturbo Operating System Technical Reference might help to understand the timing behavior of the sample application.

WARNING! The description of sample is done for OSEKturbo OS/MPC5xx v.2.2. For other platform the timing parameters may slightly differ.

Source Files

The source file that has to be changed is an main OIL file of the application that has name *main.oil*.

The content of *main.oil* file is presented below. The DS-Design Tool specific parameters are in **bold**. Note that line numbering is for explanations reasons, and line numbers should be removed before loading file into OSEK Builder.

The timing data are derived from System Service Timing where appropriate, and are in ticks. The computational timing are fictional.

The explanation of timing parameters is presented below in the listing.

```
 1 /*****************************************************************************
 2 *
 3 *       Copyright (C) 2000-2003 Motorola, Inc.
 4 *       All Rights Reserved
 5 *
 6 *       You can use this example for any purpose on any computer system with the
 7 *       following restrictions:
 8 *
 9 *       1. This example is provided "as is", without warranty.
10 *
11 *       2. You don't remove this copyright notice from this example or any direct
12 *          derivation thereof.
13 *
14 * Filename:     Source: os2000/src/osmpc/sample/standard/common/main.oil
15 * Author:       Author: bdv
16 * Locker:       Locker:
17 * State:        State: Exp
18 * Revision:     Revision: 1.6
19 *
20 * Functions:
21 *
22 * History:      Use the CVS command rlog to display revision history
23 *               information.
24 *
25 * Description:  OSEKturbo OS/MPC5xx v.2.2 sample application definitions
26 *
27 * Notes:    1. This file should be processed by OSEK system generator
28 *
29 *****************************************************************************/
30
31     APPMODE Mode{};
32     TASK TASKSND1 {
33         PRIORITY = 3;
34         SCHEDULE = FULL;
35         AUTOSTART = FALSE;
36         ACTIVATION = 1;
37         RESOURCE = MSGAACCESS;
38       ACCESSOR = SENT {
39         MESSAGE = MsgA;
40             WITHOUTCOPY = TRUE;
41             ACCESSNAME = "SND_MESS_A";
42         };
43
44         saInitial = "Init";
45         saTaskSection = SET { ID = "Init";
46             WCET = 20;
47             ACTION = GetResource { RESOURCE = MSGAACCESS; NEXT = "SendMsg"; };
```

```
   48         };
   49         saTaskSection = SET { ID = "SendMsg";
   50             WCET = 10;
   51             ACTION = SendMessage { MESSAGE = MsgA; NEXT = "MsgSent"; };
   52         };
   53         saTaskSection = SET { ID = "MsgSent";
   54             WCET = 10;
   55            ACTION = ReleaseResource { RESOURCE = MSGAACCESS; NEXT = "Terminate";
};
   56         };
   57         saTaskSection = SET { ID = "Terminate";
   58             WCET = 0;
   59         };
   60     };
   61     TASK TASKSND2 {
   62         PRIORITY = 2;
   63         SCHEDULE = FULL;
   64         AUTOSTART = FALSE;
   65         ACTIVATION = 1;
   66         RESOURCE = MSGBACCESS;
   67      ACCESSOR = SENT {
   68         MESSAGE = MsgB;
   69             WITHOUTCOPY = FALSE;
   70             ACCESSNAME = "SND_MESS_B";
   71         };
   72
   73         saInitial = "Init";
   74         saTaskSection = SET { ID = "Init";
   75             WCET = 10;
   76             ACTION = GoTo { NEXT = "Terminate"; };
   77             ACTION = GoTo { NEXT = "Send"; };
   78         };
   79         saTaskSection = SET { ID = "Send";
   80             WCET = 12;
   81             ACTION = GetResource { RESOURCE = MSGBACCESS; NEXT = "SendMsg"; };
   82         };
   83         saTaskSection = SET { ID = "SendMsg";
   84             WCET = 0;
   85             ACTION = SendMessage { MESSAGE = MsgB; NEXT = "MsgSent"; };
   86         };
   87         saTaskSection = SET { ID = "MsgSent";
   88             WCET = 0;
   89            ACTION = ReleaseResource { RESOURCE = MSGBACCESS; NEXT = "Terminate";
};
   90         };
   91         saTaskSection = SET { ID = "Terminate";
   92             WCET = 0;
   93         };
   94     };
   95     TASK TASKCNT {
   96         PRIORITY = 1;
   97         SCHEDULE = NON;
   98         AUTOSTART = FALSE;
   99         ACTIVATION = 1;
```

```
  100
  101         saTaskSection = SET { ID = "Terminate";
  102             WCET = 60;
  103         };
  104     };
  105
  106     TASK TASKRCV1 {
  107         PRIORITY = 5;
  108         SCHEDULE = FULL;
  109         AUTOSTART = TRUE;
  110         ACTIVATION = 1;
  111         RESOURCE = MSGAACCESS;
  112      ACCESSOR = RECEIVED {
  113         MESSAGE = MsgA;
  114             WITHOUTCOPY = FALSE;
  115             ACCESSNAME = "RCV_MESS_A";
  116         };
  117      EVENT = MSGAEVENT;
  118      EVENT = TIMLIMITEVENT;
  119      STACKSIZE = 400;
  120
  121         saInitial = "Init";
  122         saTaskSection = SET { ID = "Init";
  123             WCET = 270;
  124             ACTION = GoTo { NEXT = "WaitLoop"; };
  125         };
  126         saTaskSection = SET { ID = "WaitLoop";
  127             WCET = 160;
  128             ACTION = GoTo { NEXT = "WaitEvent"; };
  129         };
  130         saTaskSection = SET { ID = "WaitEvent";
  131             WCET = 0;
  132            ACTION = WaitEvent { EVENT = MSGAEVENT; EVENT = TIMLIMITEVENT; NEXT =
"CheckEvent"; };
  133         };
  134         saTaskSection = SET { ID = "CheckEvent";
  135             WCET = 12;
  136             ACTION = CheckEvent { EVENT = MSGAEVENT; SET = "GetMsg"; CLEARED =
"Init"; };
  137         };
  138         saTaskSection = SET { ID = "GetMsg";
  139             WCET = 52;
  140             ACTION = GetResource { RESOURCE = MSGAACCESS; NEXT = "MsgGot"; };
  141         };
  142         saTaskSection = SET { ID = "MsgGot";
  143             WCET = 148;
  144            ACTION = ReleaseResource { RESOURCE = MSGAACCESS; NEXT = "WaitEvent";
};
  145             DEADLINE = SET { VALUE = 20000; };
  146         };
  147     };
  148     TASK TASKRCV2 {
  149         PRIORITY = 4;
  150         SCHEDULE = FULL;
```

```
  151         AUTOSTART = FALSE;
  152         ACTIVATION = 1;
  153         RESOURCE = MSGBACCESS;
  154       ACCESSOR = RECEIVED {
  155         MESSAGE = MsgB;
  156             WITHOUTCOPY = TRUE;
  157             ACCESSNAME = "RCV_MESS_B";
  158         };
  159
  160         saInitial = "Init";
  161         saTaskSection = SET { ID = "Init";
  162             WCET = 10;
  163             ACTION = GetResource { RESOURCE = MSGBACCESS; NEXT = "MsgGot"; };
  164         };
  165         saTaskSection = SET { ID = "MsgGot";
  166             WCET = 148;
  167            ACTION = ReleaseResource { RESOURCE = MSGBACCESS; NEXT = "Terminate";
};
  168         };
  169         saTaskSection = SET { ID = "Terminate";
  170             WCET = 0;
  171             DEADLINE = SET { VALUE = 10000; };
  172         };
  173     };
  174
  175     TASK TASKSTOP {
  176         PRIORITY = 0;
  177         SCHEDULE = NON;
  178         AUTOSTART = FALSE;
  179         ACTIVATION = 1;
  180     };
  181
  182     RESOURCE MSGAACCESS {
  183         RESOURCEPROPERTY = STANDARD;
  184     };
  185     RESOURCE MSGBACCESS {
  186         RESOURCEPROPERTY = STANDARD;
  187     };
  188
  189     EVENT MSGAEVENT { MASK = AUTO; };
  190     EVENT TIMLIMITEVENT   { MASK = AUTO; };
  191
  192     COUNTER SYSTEMTIMER {
  193         MAXALLOWEDVALUE = 0xFFFFFFFF;
  194         TICKSPERBASE = 10;
  195         MINCYCLE = 0;
  196     };
  197     COUNTER SECONDTIMER {
  198         MAXALLOWEDVALUE = 30000;
  199         TICKSPERBASE = 10;
  200         MINCYCLE = 0;
  201     };
  202     COUNTER TSCOUNTER {
  203         MAXALLOWEDVALUE = 20;
```

```
  204         TICKSPERBASE = 1;
  205         MINCYCLE = 0;
  206         saPeriod = 40000; // TCOUNTER is incremented by TASKCNT, that is last in
TIMESCALE
  207     };
  208
  209     ALARM STOPALARM {
  210         AUTOSTART = FALSE;
  211         COUNTER = TSCOUNTER;
  212         ACTION = ACTIVATETASK {
  213             TASK = TASKSTOP;
  214         };
  215     };
  216     ALARM TIMLIMITALARM {
  217         AUTOSTART = FALSE;
  218         COUNTER = SECONDTIMER;
  219         ACTION = SETEVENT {
  220             TASK = TASKRCV1;
  221             EVENT = TIMLIMITEVENT;
  222         };
  223     };
  224
  225     MESSAGE MsgA {
  226         TYPE = UNQUEUED;
  227         CDATATYPE = "MSGATYPE";
  228         ACTION = SETEVENT {
  229             TASK = TASKRCV1;
  230       EVENT = MSGAEVENT;
  231        };
  232     };
  233     MESSAGE MsgB {
  234         TYPE = UNQUEUED;
  235         CDATATYPE = "MSGBTYPE";
  236         ACTION = ACTIVATETASK {
  237             TASK = TASKRCV2;
  238         };
  239     };
  240
```

There are two checkpoints in the application - reception of message *MsgA* by extended task *TASKRCV1*, and reception of message *MsgB* by basic task *TASKRCV2*. The subtransactions that ends up with these checkpoints are started by timesteps of timescale.

Therefore, timing behavior of the following system objects is defined:

- Task *TASKSND1* (lines 44-59)
- Task *TASKRCV1* (lines 121-146)

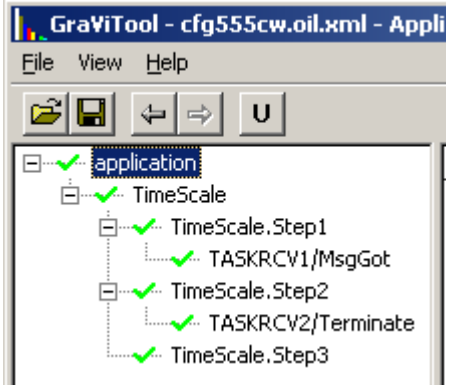
- Task *TASKSND2* (lines 73-93)
- Task *TASKRCV2* (lines 160-172)
- Task *TASKCNT* (lines 101-103)
- Counter *TSCOUNTER* (line 106)

Note that alarms are not taken into consideration as they are non-cyclic alarms, and do not severely impact application timing behavior.

The deadlines for the checkpoints are selected based on the corresponding time step. It is assumed, that message must be received until the time value of the time step expires.

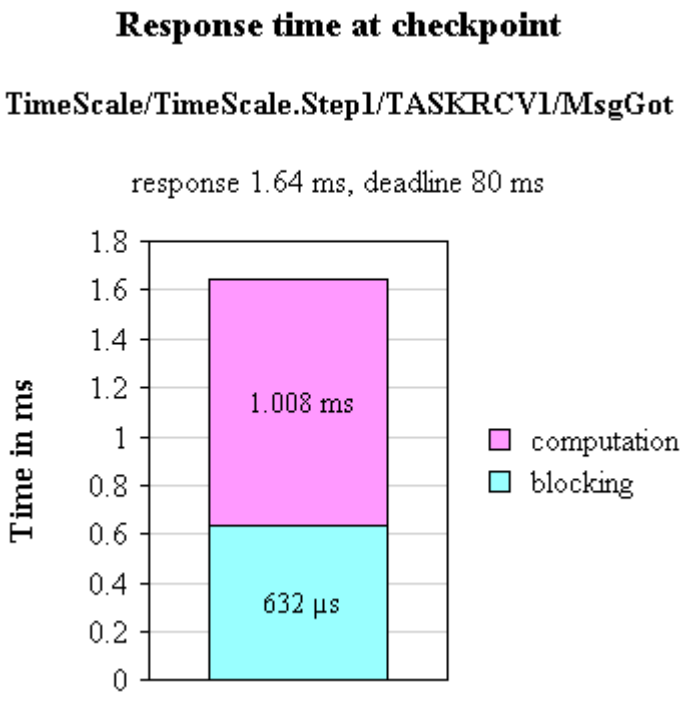
Results of Analysis

The result of schedulability analysis is as following:

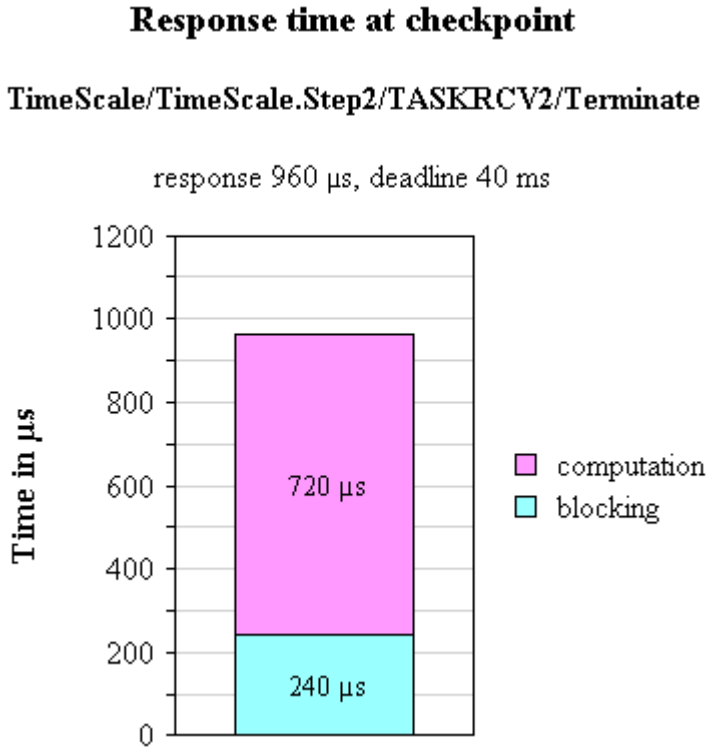


The deadlines are met.

Message *MsgA* is received within deadline though response time experiences blocking from resources manipulation to protect *MsgB*:



Message *MsgB* is also received within deadline though blocking from *TASKCNT* takes place:



B

Error Messages

The Schedulability Analysis for OSEKturbo checks the compatibility of properties, parameters and limits and reports about possible errors via error messages. The error messages can be associated with wrong syntax, mistakes in the implementation definition, wrong definitions of the application objects.

This chapter contains only the list of System Generator message specific to DS-Design Tool. Description of System Generator error processing can be found in OSEKturbo User's Manuals:

- List of System Generator Messages

List of System Generator Messages

SysGen consists of several components, which work on the different stages of OIL file processing. Schedulability analysis is performed by DS-Design Tool component, and generates DS-Design Tool specific messages having prefix **SA** before the message number.

SA0005: **Could not create Msxml2.MXXMLWriter.4.0 coclass (MSXML4 is not installed?)**
`Error`

Output file cannot be created because MSXML4 is not installed on the computer. This free downloadable software is used to generate auxiliary DS-Design Tool files. In case the message appears, check System Generator options, or download the software.

SA0006: **Could not open file <filename> for writing**
`Fatal`

The file <filename> cannot be opened if there is no enough free disk space or there is no appropriate permission to create or write the output file.

SA0008: **Output GView file is not specified. <filename> will be used**
`Warning`

There is no file name specified in -z option of System Generator. DS-Design Tool will use <filename> for graphical views output.

SA0010: Duplicated identifier: <identifier>
`Error`

The identifier <identifier> is used more than once for different purposes (e.g. there are two task sections with same ID value).

SA0012: Invalid initial section: <section>. No such section exists
`Error`

The task or the ISR section referenced in *saInitial* parameter does not exist.

SA0013: Initial section must be specified
`Error`

There is no *saInitial* section referenced in the task or in the ISR. If the task or the ISR consists of more than one section, than initial (start) section must be referenced in *saInitial* parameter of the *TASK* or *ISR*.

SA0021: Referenced resource <resource> is not accessed by the task (ISR)
`Error`

Resource specified in *ACTION* parameter is not referenced in *TASK* or *ISR* parameters.

SA0023: Invalid reference to <object>. No such object exists
`Error`

The object to which application references, does not exist (e.g., task section referenced in *NEXT* parameter, does not exist).

SA0025: Action "<action>" not permitted in context of ISR of category 1
`Error`

The action <action> is not allowed in ISR of category 1 (e.g., ISR of category should not contain action *ActivateTask*)

SA0026: Counter period is already specified (or calculated)
`Error`

The period of the counter is already specified. (e.g., application specifies *PERIOD* parameter of a the counter, that is linked to system timer. As system timer period is calculated automatically, parameter *PERIOD* in *COUNTER* object is redundant and ambiguous).

SA0027: **Sorry, the feature is not supported yet**
`Error`

The feature schedulability analysis is not supported in this version of DS-Design Tool. (e.g. the action *SetRelAlarm* is not supported as DS-Design Tool does not analyze dynamic alarms).

SA0028: **Counter <counter> is unreferenced, its period cannot be evaluated**
`Error`

The period of the counter cannot be evaluated because there is no references to it (e.g. from system timer, and the *COUNTER* object has no parameter *PERIOD* specified).

SA0029: **Priority must be not greater than <maxpriority>**
`Error`

The priority of a system object is too high.

SA0031: **Ignored one-shot alarm <alarm>**
`Warning`

Alarm <alarm> is not cyclic, and is ignored by DS-Design Tool. This may happen when *saCyclic* parameter in *ALARM* object is set to *FALSE*.

SA0032: **ISR must have positive minimal inter-arrival time**
`Error`

The *saMinIATime* parameter in *ISR* object is zero. Minimal interarrival time of an ISR shall have positive value.

SA0033: **Ignored one-shot ISR <ISR>**
`Warning`

An ISR does not have saMinIATime specified, and is ignored therefore.

SA0037: **Array of EVENTs must contain at least one event**
`Error`

There is not any event name specified in actions where list of events is accepted (e.g. action *SetEvent* specifies *TASK* parameter, but does not has any *EVENT* parameter).

SA0038: **Attempt to wait for non-owned events [<event>]**
`Error`

The task section contains an action *WaitEvent* with *EVENT* parameter value that it is not owned by the task.

SA0039: Multiple reference to event <event> in list
`Warning`

Same event is referenced more than once in a list of events (e.g. *WaitEvent* action contains two values of same event name).

SA0040: Execution graph is inconsistent. Sections [<section>] cannot be reached
`Error`

The task or an ISR section can not be reached as there is no link to it from any other sections, and it is not specified as *saInitial* section (e.g. task section is skipped by mistake).

SA0041: Invalid call in DisableAllInterrupts context
`Error`

The prohibited system service is called when all interrupts are disabled. The only allowed call is *EnableAllInterrupts.*

SA0042: Invalid call in SupendAllInterrupts/SuspendOSInterrupts context
`Error`

The prohibited system service is called within pairs of *SuspendAllInterrupts/ResumeAllInterrupts* and *SuspendOSInterrupt/ ResumeOSInterrupts*. The rules on allowed services are specified in OSEKturbo Operating System User's Manual.

SA0043: Invalid call within critical section
`Error`

The prohibited system service is called within critical section framed by *GetResource/ReleaseResource* pairs (e.g. system service *Schedule* is called within critical section).

SA0044: GetResource/ReleaseResource mismatch
`Error`

The critical sections framed by *GetResource/ReleaseResource* pairs are not consistent (e.g. task gets first resource, and releases second, or task gets first and second resources, and releases them in non-stacked order).

SA0045: Attempt to set events [<event>] not owned by TASK <task>

```
Error
```

The *SetEvent* action in the task or the ISR section sets the event for the task when this task is not an owner of the event.

SA0046: Multiple activation of extended task <task>
```
Error
```

The extended task <task> gets activated more than once (e.g. the task is autoactivated, and also explicitly activated in an ISR).

SA0047: Attempt to set event for suspended task <task>
```
Warning
```

The action in a task or in an ISR section sets the event(s) for the extended task that has never been activated (e.g. the task is not autostarted, no any section activates or chains this task).

SA0050: Extended task <task> may be activated by different AUTOSTART tasks
```
Error
```

The extended task <task> is activated by more than one task that start automatically (e.g. two autostarted task chain same extended, or extended task starts automatically, and is also activated by another task).

SA0051: Execution paths of autostart task <task> form different extended tasks context
```
Error
```

The autostarted task <task> is split into paths that lead to different contexts (e.g. autostarted task has an initial section with two *ChainTask* actions for two different paths).

SA0052: Attempt to activate task <task> with undefined structure
```
Error
```

The activated task <task> has no DS-Design Tool specific parameters, and the timing behavior is not defined therefore (e.g. an ISR that has timing information described in ISR sections, activates the task that has no sections at all).

SA0054: Extended tasks state is changed on subtransaction completion
```
Error
```

The extended task is changing it state in a course of subtransaction (e.g. extended task handles an event that is set, and then terminates itself, thus changing it state to *suspended*).

SA0055: **Unexpected interrupt control service (mismatch)**
`Error`

The Fast Disable/Enable API services are messed up (e.g. *SuspendOSInterrupts* is followed by *ResumeAllInterrupts*).

SA0056: **Attempt to get already occupied resource <resource>**
`Error`

The task or the ISR attempts to get same resource <resource> twice without releasing it first.

SA0058: **Subtransaction defined with an action void and shall be ignored**
`Warning`

Void action of subtransaction makes subtransaction invalid.

SA0059: **Could not build subtransaction <subtransaction> profile. Check if loop presents**
`Error`

Subtransaction possibly goes into loop on the execution graph.

SA0065: **Clock frequency is either not specified or zero**
`Error`

Clock frequency shall be defined explicitly.

SA0066: **Clock frequency differs from one specified in TargetMCU section. This new value shall be used**
`Warning`

ClockFrequency attribute is specified in TargetMCU section. The attribute saClockFrequency is differs from it. The value of saClockFrequency shall be used.

SA0067: **Cannot properly scale timing values. Analysis results are invalid**
`Warning`

Certain processors (e.g. MGT5x00) are complicated enough, and increasing of oscillator frequency does not lead to proportional decreasing of execution times. Thus special invalidation flag saInvalidate controls timing adjustment. If this flag is set to TRUE DS-Design Tool will warn

user about potential invalid anaylsis results in case when current oscillator frequency differs from the one at which execution times were measured.

SA0068: **TASK <task> may not be executed due to its effective utilization (XXX.YY%) exceeds 1**
ISR <ISR> may not be executed due to its effective utilization (XXX.YY%) exceeds 1
`Warning`

The task or ISR specified has no checkpoints and thus analysis is not performed for it. However, effective utilization for this task/ISR exceeds 1, which may lead to occasional or permanent overflow (i.e. loss of activation request or interrupt occurence). The user has to decide if this is acceptable and won't break overall system behavior, and has to take appropriate measures (if this is very undesirable) or just ignore it (if this leads, for instance, just to reasonable performance degradation).

SA0100: **Analysis results: application is schedulable**
`Information`

Verdict on application schedulability.

SA0101: **Analysis results: application is not schedulable**
`Warning`

Verdict on application schedulability.

SA0102: **Analysis results: no checkpoints specified**
`Information`

Verdict on application schedulability.

SA0103: **Checkpoint <checkpoint>: OVERFLOW. Utilization is XX.YY%**
`Warning`

Checkpoint <checkpoint> is not computed because deadline can not be met due to overutilization.

SA0104: **Checkpoint <checkpoint>: response is XXX, deadline is YYY.**
`Information`

Checkpoint <checkpoint> meets deadline.

SA0105: **Checkpoint <checkpoint>: response is XXX, deadline is YYY. NOT SCHEDULABLE**

Warning

Checkpoint <checkpoint> misses deadline.

SA0106: Total utilization is XXX.YY%
Warning

Total utilization is 100% and above.

SA0107: Total utilization is XX.YY%
Information

Total utilization is below 100%.

SA0108: Analysis results: application is schedulable (all deadlines are met) though total utilization exceeds 1.
Warning

Effective utilization of CPU for the aplication exceeds 1, which may lead to occasional or permanent overflow (i.e. loss of activation request or interrupt occurence). The user has to decide if this is acceptable and won't break overall system behavior, and has to take appropriate measures (if this is very undesirable) or just ignore it (if this leads, for instance, just to reasonable performance degradation).

C

OIL Language Quick Reference

This chapter consists of the following sections:

- OS Object
- TASK Object
- ISR Object
- RESOURCE Object
- EVENT Object
- COUNTER Object
- ALARM Object
- MESSAGE Object
- APPMODE Object
- COM Object
- NM Object

The lists of all OIL object parameters that are DS-Design Tool specific. Parameters with their possible values and short descriptions are provided here.

The value used by default is typed in boldface in the *Possible Values* cells.

NOTE Standard parameters and OSEKturbo specific parameters are described in OSEKturbo Operating System documentation, and are duplicated here only if DS-Design Tool makes special use of the standard or OSEKturbo specific parameters.

NOTE All DS-Design Tool specific parameters are also OSEKturbo specific as currently no standard OIL attributes are defined for schedulability analysis tools.

OS Object

The brief description of Operating System global parameters.

Table C.1 OS Parameters

Object Parameters	Possible Values	Description
DS-Design Tool specific attributes		

The attributes should be defined inside the scope of the OS object in accordance with the following syntax[a]:
`saTimeUnit = < ns / us / ms / s>;`
`saClockFrequency = <integer>;`
`saMeasureClockFrequency = <integer>;`
`saInvalidate = <TRUE/FALSE>;`
`saOSOverhead = <SET> {`
`  <OS Overhead Parameters>`
`};`		

saTimeUnit	enum of ns, **us**, ms, s	Units used by DS-Design Tool for all time measurement parameters: ns - nanoseconds, us - microseconds, ms - milliseconds; s - seconds
saClockFrequency	integer	Current frequency of oscillator (same units as TargetMCU's ClockFrequency). If ClockFrequency is not set, then saClockFrequency should be set in kHz units
saMeasureClockFrequency	integer	Frequency of oscillator used when measurement of timing parameters was done. This value should not be changed. If value of saClockFrequency is not equal to the value of saMeasureClockFrequency, then the values of frequency-dependent parameters (OS overheads and WCET) are proportionally adjusted.
saInvalidate	TRUE **FALSE**	If set to TRUE, the warning is generated by DS-Design Tool if the values of saClockFrequency and saMeasureClockFrequency are not equal. This is to inform the user that automatic adjustment is performed, but processor complexity does not guarantee the adjustment is valid.
saOSOverhead		Timing of Operating System services needed for proper calculation
DEBUG_LEVEL	4	Specifies ORTI support that includes trace capabilities for DS-V

[a] Operating System overhead parameters are dedicated to OSEKturbo developers, and are listed in **Table C.2** for information only

The brief description of Operating System global parameters. Most parameters specify Operating System overhead for performing system services.

Table C.2 OS Overhead Parameters

Object Parameters	Possible Values	Description
DS-Design Tool specific attributes		
SA_ACTIVATE_TASK_PREEMPT	integer, **0**	OS overhead to perform system service *ActivateTask* to activate new task and to preempt running task (dispatching from the calling task)
SA_ACTIVATE_TASK_NO_PREEMPT	integer, **0**	OS overhead to perform system service *ActivateTask* to activate new task without the preemption of the running task (no dispatching from the calling task)
SA_ISR_ACTIVATE_TASK	integer, **0**	OS overhead to perform system service *ActivateTask* within an ISR to activate new task
SA_TERMINATE_TASK_DISPATCH	integer, **0**	OS overhead to perform system service *TerminateTask* to terminate the running task and to dispatch to the ready task
SA_CHAIN_TASK_DISPATCH	integer, **0**	OS overhead to perform system service *ChainTask* to chain the new task, to terminate the running task, and to dispatch to the ready task
SA_SCHEDULE_PREEMPT	integer, **0**	OS overhead to perform system service *Schedule* to call scheduler and to preempt the running task (dispatching from the calling task)
SA_SCHEDULE_NO_PREEMPT	integer, **0**	OS overhead to perform system service *Schedule* to call scheduler without the preemption of the running task (no dispatching from the calling task)
SA_DISABLE_ALL_INTERRUPTS	integer, **0**	OS overhead to perform system service *DisableAllInterrupts*

Table C.2 OS Overhead Parameters

SA_ENABLE_ALL_INTERRUPTS	integer, **0**	OS overhead to perform system service *EnableAllInterrupts*
SA_SUSPEND_ALL_INTERRUPTS	integer, **0**	OS overhead to perform system service *SuspendAllInterrupts*
SA_RESUME_ALL_INTERRUPTS	integer, **0**	OS overhead to perform system service *ResumeAllInterrupts*
SA_SUSPEND_OS_INTERRUPTS	integer, **0**	OS overhead to perform system service *SuspendOSInterrupts*
SA_RESUME_OS_INTERRUPTS	integer, **0**	OS overhead to perform system service *ResumeOSInterrupts*
SA_GET_RESOURCE	integer, **0**	OS overhead to perform system service *GetResource* to acquire the resource
SA_ISR_GET_RESOURCE	integer, **0**	OS overhead to perform system service *GetResource* within an ISR to acquire the resource
SA_RELEASE_RESOURCE_PREEMPT	integer, **0**	OS overhead to perform system service *ReleaseResource* to release the resource and to preempt the running task (dispatching from the calling task)
SA_RELEASE_RESOURCE_NO_PREEMPT	integer, **0**	OS overhead to perform system service *ReleaseResource* to release the resource without preemption of the running task (no dispatching from the calling task)
SA_ISR_RELEASE_RESOURCE	integer, **0**	OS overhead to perform system service *ReleaseResource* within an ISR to release the resource
SA_SET_EVENT_PREEMPT	integer, **0**	OS overhead to perform system service *SetEvent* to set the event(s) and to preempt the running task (dispatching from the calling task)

Table C.2 OS Overhead Parameters

SA_SET_EVENT_NO_PREEMPT	integer, **0**	OS overhead to perform system service *SetEvent* to set the event(s) without preemption of the running task (no dispatching from the calling task)
SA_ISR_SET_EVENT	integer, **0**	OS overhead to perform system service *SetEvent* within an ISR to set the event(s)
SA_WAIT_EVENT_PREEMPT	integer, **0**	OS overhead to perform system service *WaitEvent* to wait the event(s) and to preempt the running task (dispatching from the calling task)
SA_WAIT_EVENT_NO_PREEMPT	integer, **0**	OS overhead to perform system service *WaitEvent* to wait the event(s) without preemption of the running task (no dispatching from the calling task)
SA_SEND_MESSAGE_ACTIVATE_TASK_PREEMPT	integer, **0**	OS overhead to perform system service *SendMessage* to send the message, to activate the task, and to preempt the running task (dispatching from the calling task)
SA_SEND_MESSAGE_ACTIVATE_TASK_NO_PREEMPT	integer, **0**	OS overhead to perform system service *SendMessage* to send the message, to activate the task without preemption of the running task (no dispatching from the calling task)
SA_SEND_MESSAGE_SET_EVENT_PREEMPT	integer, **0**	OS overhead to perform system service *SendMessage* to send the message, to set the event, and to preempt the running task (dispatching from the calling task)
SA_SEND_MESSAGE_SET_EVENT_NO_PREEMPT	integer, **0**	OS overhead to perform system service *SendMessage* to send the message, to send the event without preemption of the running task (no dispatching from the calling task)
SA_ISR_SEND_MESSAGE_ACTIVATE_TASK	integer, **0**	OS overhead to perform system service *SendMessage* within an ISR to send the message and to activate the task

Table C.2 OS Overhead Parameters

SA_ISR_SEND_MESSAGE_SET_EVENT	integer, **0**	OS overhead to perform system service *SendMessage* within an ISR to send the message and to set the event
SA_ENTER_ISR	integer, **0**	OS overhead to perform system service *EnterSR*
SA_LEAVE_ISR	integer, **0**	OS overhead to perform system service *LeaveISR*
SA_ALARM_ACTIVATETASK	integer, **0**	OS overhead to perform activation of the task when an alarm expires
SA_ALARM_SETEVENT	integer, **0**	OS overhead to perform set of the event when an alarm expires
SA_TIMESCALE_ACTIVATE	integer, **0**	Virtual OS overhead to perform task activation within the task scale

TASK Object

The brief description of task parameters.

Table C.3 TASK Parameters

Object Parameters	Possible Values	Description
DS-Design Tool specific attributes		

The attributes should be defined inside the scope of the TASK object in accordance with the following syntax:
`saInitial = <string>;`
`saTaskSection = <SET> {`
`  <task section>`
`};`		

saInitial	string, ““	Name of first task section in the execution graph. If task consists of no more than one task section, the default value is empty. If task consists of more than one task sections, there is no default value
saTaskSection	set	Task section (see **Table C.4**)

The brief description of task section parameters.

Table C.4 saTaskSection Parameters

Object Parameters	Possible Values	Description
DS-Design Tool specific attributes		

The attributes should be defined inside the scope of the TASK object in accordance with the following syntax:
`saTaskSection = <SET> {`
`  ID = <string>;`
`  WCET = <integer>;`
`  DEADLINE[a] = <SET> {`
`  <deadline of task section>`
`  };`
`  ACTION[b] = <name of ACTION> {`
`  <action of task section>`
`  };`
`};`		

ID	string	Name of this task section
WCET	integer, **0**	Worst-case execution time of task section
DEADLINE	set	Specification of the deadline of the task section (see **Table C.5**)
ACTION	ActivateTask, **TerminateTask**, ChainTask, Schedule, DisableAllInterrupts, EnableAllInterrupts, SuspendAllInterrupts, ResumeAllInterrupts, SuspendOSInterrupts, ResumeOSInterrupts, GetResource, ReleaseResource, SetEvent, WaitEvent, CheckEvent, SendMessage, GoTo	Action that ends the task section (see **Table C.6**)

a. Multiply instances of DEADLINE are allowed in one task section

b. Multiply instances of ACTION are allowed in one task section

The brief description of task section's deadline parameters

Table C.5 saTaskSection/DEADLINE Parameters

Object Parameters	Possible Values	Description
DS-Design Tool specific attributes		

The attributes should be defined inside the scope of the saTaskSection of the TASK object in accordance with the following syntax:
`DEADLINE = <SET> {`
`  VALUE = <integer>;`
`  PATH = <string>;`
`};`		

VALUE	integer	Deadline value relative to the start of the subtransaction
PATH	string	Path to which this deadline of the task section belongs

The brief description of task section's action parameters

Table C.6 saTaskSection/ACTION Parameters

Object Parameters	Possible Values	Description
DS-Design Tool specific attributes		

The attributes should be defined inside the scope of the saTaskSection of the TASK object in accordance with the following syntax:
`ACTION = ActivateTask {`
`  NEXT = <string>;`
`  TASK = <name of TASK>;`
`  PATH = <string>;`
`};`
`ACTION = TerminateTask {`
`  PATH = <string>;`
`};`
`ACTION = ChainTask {`
`  TASK = <name of TASK>;`
`  PATH = <string>;`
`};`
`ACTION = Schedule {`
`  NEXT = <string>;`
`  PATH = <string>;`
`};`		

Table C.6 saTaskSection/ACTION Parameters

```
ACTION = <DisableAllInterrupts / EnableAllInterrupts> {
  NEXT = <string>;
  PATH = <string>;
};
ACTION = <SuspendAllInterrupts / ResumeAllInterrupts> {
  NEXT = <string>;
  PATH = <string>;
};
ACTION = <SuspendOSInterrupts / ResumeOSInterrupts> {
  NEXT = <string>;
  PATH = <string>;
};
ACTION = <GetResource / ReleaseResource> {
  NEXT = <string>;
  RESOURCE = <name of RESOURCE>;
  PATH = <string>;
};
ACTION = SetEvent {
  NEXT = <string>;
  TASK = <name of TASK>;
  EVENT = <name of EVENT>;
  PATH = <string>;
};
ACTION = WaitEvent {
  NEXT = <string>;
  EVENT = <name of EVENT>;
  PATH = <string>;
};
ACTION = CheckEvent {
  EVENT = <name of EVENT>;
  SET = <string>;
  CLEARED = <string>;
  PATH = <string>;
};
```

Table C.6 saTaskSection/ACTION Parameters

```
ACTION = SendMessage {
  NEXT = <string>;
  MESSAGE = <name of MESSAGE>;
  PATH = <string>;
};

ACTION = GoTo {
  NEXT = <string>;
  PATH = <string>;
};
```

ACTION	ActivateTask	Task section calls OS service *ActivateTask*
	TerminateTask	Task section calls OS service *TerminateTask*. This is default action.
	ChainTask	Task section calls OS service *ChainTask*
	Schedule	Task section calls OS service *Schedule*
	DisableAllInterrupts	Task section calls OS service DisableAllInterrupts
	EnableAllInterrupts	Task section calls OS service EnableAllInterrupts
	SuspendAllInterrupts	Task section calls OS service SuspendAllInterrupts
	ResumeAllInterrupts	Task section calls OS service ResumeAllInterrupts
	SuspendOSInterrupts	Task section calls OS service SuspendOSInterrupts
	ResumeOSInterrupts	Task section calls OS service ResumeOSInterrupts
	GetResource	Task section calls OS service *GetResource*
	ReleaseResource	Task section calls OS service *ReleaseResource*
	SetEvent	Task section calls OS service *SetEvent*
	WaitEvent	Task section calls OS service *WaitEvent*
	CheckEvent	Task section checks the state of an event and jumps to the *NEXT* task section depending on the state of the event
	SendMessage	Task section calls OS service *SendMessage*
	GoTo	Task section jumps to the *NEXT* task section
PATH	string, ““	Path to which this action of the task section belong. This parameter is applicable for all actions
NEXT	string	Reference (link) to the next task section in the subtransaction. Section ID is used as reference. This parameter is applicable for all actions **except** *TerminateTask, ChainTask*, and *CheckEvent*

Table C.6 saTaskSection/ACTION Parameters

TASK	name of TASK	Reference to the task that is used in this action. This parameter is applicable **only** for actions *ActivateTask*, *ChainTask*, and *SetEvent*
RESOURCE	name of RESOURCE	Reference to the resource that is used in this action. This parameter is applicable **only** for actions *GetResource* and *ReleaseResource*
EVENT	name of EVENT	Reference to the list of events that are used in this action. This parameter is applicable **only** for actions *SetEvent* and *WaitEvent*
EVENT	name of EVENT	Reference to the event that is used in this action. This parameter is applicable **only** for action *CheckEvent*
SET	string	Reference (link) to the next task section in the subtransaction. The next task section is executed if the event is set. Section ID is used as reference. This parameter is applicable **only** for action *CheckEvent*
CLEARED	string	Reference (link) to the next task section in the subtransaction. The next task section is executed if the event is cleared (event is not set). Section ID is used as reference. This parameter is applicable **only** for action *CheckEvent*
MESSAGE	name of MESSAGE	Reference to the message that is used in this action. This parameter is applicable **only** for actions *SendMessage*

ISR Object

The brief description of interrupt service routine parameters.

Table C.7 ISR Parameters

Object Parameters	Possible Values	Description
DS-Design Tool specific attributes		

The attributes should be defined inside the scope of the ISR object in accordance with the following syntax:
`saInitial = <string>;`
`saMinIATime = <integer>;`
`saJitter = <string>;`
`saISRSection = <SET> {`
`  <ISR section>`
`};`		

saInitial	string, ““	Name of first ISR section in the execution graph. If ISR consists of no more than one ISR section, the default value is empty. If ISR consists of more than one ISR sections, there is no default value

Table C.7 ISR Parameters

saISRSection	set	ISR section (see **Table C.8**)
saMinIATime	integer, **0**	ISR minimal interarrival time
saJitter	integer, **0**	ISR release jitter

The brief description of interrupt service routine section parameters.

Table C.8 saISRSection Parameters

Object Parameters	Possible Values	Description
DS-Design Tool specific attributes		

The attributes should be defined inside the scope of the ISR object in accordance with the following syntax:
`saISRSection = <SET> {`
`  ID = <string>;`
`  WCET = <integer>;`
`  DEADLINE`[a] `= <SET> {`
`  <deadline of ISR section>`
`  };`
`  ACTION`[b] `= <name of ACTION> {`
`  <action of ISR section>`
`  };`
`};`		

ID	string	Name of this ISR section
WCET	integer	Worst-case execution time of the ISR section
DEADLINE	set	Specification of deadline of the ISR section (see **Table C.9**)
ACTION	ActivateTask, DisableAllInterrupts, EnableAllInterrupts, SuspendAllInterrupts, ResumeAllInterrupts, SuspendOSInterrupts, ResumeOSInterrupts, GetResource, ReleaseResource, SetEvent, SendMessage, GoTo, **LeaveISR**	Action that ends the ISR section (see **Table C.10**)

a. Multiply instances of DEADLINE are allowed in one ISR section

b. Multiply instances of ACTION are allowed in one ISR section

The brief description of ISR section's deadline parameters

Table C.9 saISRSection/DEADLINE Parameters

Object Parameters	Possible Values	Description
DS-Design Tool specific attributes		

The attributes should be defined inside the scope of the saISRSection of the ISR object in accordance with the following syntax:
`DEADLINE = <SET> {`
`  VALUE = <integer>;`
`  PATH = <string>;`
`};`		

VALUE	integer	Deadline value relative to the start of the subtransaction
PATH	string	Path to which this deadline of the ISR section belongs

The brief description of ISR section's action parameters

Table C.10 saISRSection/ACTION Parameters

Object Parameters	Possible Values	Description
DS-Design Tool specific attributes		

The attributes should be defined inside the scope of the saTaskSection of the TASK object in accordance with the following syntax:
`ACTION = ActivateTask {`
`  NEXT = <string>;`
`  TASK = <name of TASK>;`
`  PATH = <string>;`
`};`
`ACTION = <DisableAllInterrupts / EnableAllInterrupts> {`
`  NEXT = <string>;`
`  PATH = <string>;`
`};`
`ACTION = <SuspendAllInterrupts / ResumeAllInterrupts> {`
`  NEXT = <string>;`
`  PATH = <string>;`
`};`		

Table C.10 salSRSection/ACTION Parameters

```
ACTION = <SuspendOSInterrupts / ResumeOSInterrupts> {
  NEXT = <string>;
  PATH = <string>;
};

ACTION = <GetResource / ReleaseResource> {
  NEXT = <string>;
  RESOURCE = <name of RESOURCE>;
  PATH = <string>;
};

ACTION = SetEvent {
  NEXT = <string>;
  TASK = <name of TASK>;
  EVENT = <name of EVENT>;
  PATH = <string>;
};

ACTION = SendMessage {
  NEXT = <string>;
  MESSAGE = <name of MESSAGE>;
  PATH = <string>;
};

ACTION = GoTo {
  NEXT = <string>;
  PATH = <string>;
};

ACTION = LeaveISR {
  PATH = <string>;
};
```

Table C.10 saISRSection/ACTION Parameters

ACTION	ActivateTask	ISR section calls OS service *ActivateTask*
	DisableAllInterrupts	ISR section calls OS service DisableAllInterrupts
	EnableAllInterrupts	ISR section calls OS service EnableAllInterrupts
	SuspendAllInterrupts	ISR section calls OS service SuspendAllInterrupts
	ResumeAllInterrupts	ISR section calls OS service ResumeAllInterrupts
	SuspendOSInterrupts	ISR section calls OS service SuspendOSInterrupts
	ResumeOSInterrupts	ISR section calls OS service ResumeOSInterrupts
	GetResource	ISR section calls OS service *GetResource*
	ReleaseResource	ISR section calls OS service *ReleaseResource*
	SetEvent	ISR section calls OS service *SetEvent*
	SendMessage	ISR section calls OS service *SendMessage*
	GoTo	ISR section jumps to the *NEXT* ISR section
	LeaveISR	ISR section calls OS service *LeaveISR*. This is default action.
PATH	string	Path to which this action of the ISR section belongs
NEXT	string	Reference (link) to the next ISR section in the subtransaction. Section ID is used as reference. This parameter is applicable for all actions **except** *LeaveISR*
TASK	name of TASK	Reference to the task that is used in this action. This parameter is applicable **only** for action *ActivateTask* and *SetEvent*
RESOURCE	name of RESOURCE	Reference to the resource that is used in this action. This parameter is applicable **only** for actions *GetResource* and *ReleaseResource*
EVENT	name of EVENT	Reference to the list of events that are used in this action. This parameter is applicable **only** for action *SetEvent*
MESSAGE	name of MESSAGE	Reference to the message that is used in this action. This parameter is applicable **only** for actions *SendMessage*

RESOURCE Object

There is no DS-Design Tool specific parameters.

EVENT Object

There is no DS-Design Tool specific parameters.

COUNTER Object

The brief description of counter parameters.

Table C.11 COUNTER Parameters

Object Parameters	Possible Values	Description
DS-Design Tool specific attributes		
The attributes should be defined inside the scope of the COUNTER object in accordance with the following syntax: `saPeriod = <integer>;`		
saPeriod	integer, **0**	Period of the counter

ALARM Object

The brief description of alarm parameters.

Table C.12 ALARM Parameters

Object Parameters	Possible Values	Description
DS-Design Tool specific attributes		

The attributes should be defined inside the scope of the ALARM object in accordance with the following syntax:
`saCyclic = <TRUE / FALSE> {`
`  ALARMTIME = <integer>;`
`  CYCLETIME = <integer>;`
`};`		

saCyclic	TRUE, **FALSE**	TRUE, if alarm is cyclic; FALSE otherwise
ALARMTIME	integer, **0**	Alarm offset (if alarm is cyclic). Measured in timer counter ticks
CYCLETIME	integer	Alarm cycle (if alarm is cyclic). Measured in timer counter ticks

MESSAGE Object

There is no DS-Design Tool specific parameters.

APPMODE Object

There is no DS-Design Tool specific parameters.

COM Object

There is no DS-Design Tool specific parameters.

NM Object

There is no DS-Design Tool specific parameters.

Index

A

action
    Task or ISR section 52

B

blocking pessimism 170

C

checkpoint 62
computation time 16
computational model 19
computational task 17

D

deadline 16
    Task or ISR section 52
Deadline Monotonic Algorithm 24
DMA 24

E

execution graph 54, 55
execution scenario 54

F

fixed-priority preemptive scheduling 18
fixed-priority scheduling 18
full-preemptive scheduling 18

G

graphical views file 28
Graphical Views Tool 28

H

hard deadline 16
hard real-time application 17

I

ISR section 51

M

minimal interarrival time 48

N

next
    Task or ISR section 54

O

OIL parameters
    ALARM
        ALARMTIME 204
        CYCLETIME 204
        CYCLIC 204
    COUNTER
        Period 204
    ISR
        INITIAL 199
        ISR_SECTION 200
            ACTION 200
                ActivateTask 201
                DisableAllInterrupts 201
                EnableAllInterrupts 201
                EVENT 203
                GetResource 202
                Goto 202
                LeaveISR 202
                MESSAGE 203
                NEXT 203
                PATH 203
                ReleaseResource 202
                RESOURCE 203
                ResumeAllInterrupts 201
                ResumeOSInterrupts 202
                SendMessage 202
                SetEvent 202
                SuspendAllInterrupts 201
                SuspendOSInterrupts 202
                TASK 203
            DEADLINE 200

For More Information: www.freescale.com

PATH 201
VALUE 201
ID 200
WCET 200
JITTER 200
MIN_IA_TIME 200
OS
DEBUG_LEVEL 190
SATimeUnit 190
TASK
INITIAL 194
TASK_SECTION
ACTION 195
ActivateTask 196
ChainTask 196
CheckEvent 197
CLEARED 199
DisableAllInterrupts 197
EnableAllInterrupt 197
EVENT 199
GetResource 197
Goto 198
MESSAGE 199
NEXT 198
PATH 198
ReleaseResource 197
RESOURCE 199
ResumeAllInterrupt 197
ResumeOSInterrupt 197
Schedule 196
SendMessage 198
SET 199
SetEvent 197
SuspendAllInterrupts 197
SuspendOSInterrupts 197
TASK 199
TerminateTask 196
WaitEvent 197
DEADLINE 195
PATH 196
VALUE 196
ID 195
WCET 195

P

periodic transaction 48
preemptive dispatching 18

R

rate monotonic algorithm 23
real time 15
real-time computer system 15
realtime in operating systems 15
real-time systems 15
release jitter 48
response time 16
RMA 24

S

SA 19
schedulability analysis 19
scheduling with preemption threshold 159
soft deadline 17
sporadic transaction 48
subtransactions approach 25

T

task section 51
text views 29, 160
timepoint 62
transaction 46
transaction concept 24

U

utilization bound 23

V

V-shaped development process 21

W

WCET 19
worst-case execution time 19